McGraw Hill
Michael L. Schmit
Il Manuale Pentium
Strumenti di ottimizzazione
informatica professionale

Informatica professionale

Indice

PARTE TERZA • INTRODUZIONE AL PENTIUM E AI RELATIVI STRUMENTI DI SVILUPPO

Prefazione

Questo libro è dedicato ai programmatori che desiderano apprendere le tecniche di ottimizzazione avanzata dei programmi per la famiglia di microprocessori Intel 80x86, incluso il Pentium.

I primi capitoli presentano le basi della programmazione in linguaggio assembler 80x86 ma per comprendere gli ultimi capitoli è necessaria una conoscenza media della programmazione assembler o C.

Che cosa si trova sul disco

Il disco include tutti i listati presentati in questo manuale, una versione del programma Quantasm PentOpt (un ottimizzatore per Pentium) e DEBUG32 (un debugger DPMI per programmi a 32 bit operanti in modalità protetta). La maggior parte del codice presentato può funzionare con gli assembler Microsoft MASM 5.1 o successivi (preferibilmente a partire dal MASM 6.0) oppure Borland TASM (qualsiasi versione). Alcuni degli esempi riguardanti il Pentium richiedono l'impiego dell'assembler MASM 6.11, il primo assembler Microsoft a essere dotato del supporto per il Pentium. Quest'ultimo requisito può essere ignorato scrivendo le macro che generano i codici operativi corretti.

Perché è importante apprendere l'uso del linguaggio assembler per il Pentium

Il mondo del software è sempre più rivolto alla programmazione orientata agli oggetti e ai linguaggi di alto livello con sistemi operativi sempre più complessi. Dunque ci si potrebbe chiedere il motivo per il quale ci si dovrebbe occupare del linguaggio assembler per il Pentium. La risposta è semplice: la velocità. Vi possono essere anche altri motivi che spingono a programmare in linguaggio assembler ma certamente il motivo più importante è la velocità. Se questo non fosse un problema, si potrebbe ancora pro-

grammare con il linguaggio assembler per l'8088. Il Pentium contiene due pipeline per l'esecuzione di istruzioni, entrambe equivalenti a quelle presenti nelle CPU 386 o 486. Il problema è proprio quello di mantenere occupate il più possibile entrambe le pipeline.

Se si programma perlopiù in un linguaggio di alto livello, come ad esempio il C/C++, la conoscenza del linguaggio assembler può aiutare a comprendere il linguaggio, i suoi aspetti meno evidenti e le prestazioni raggiungibili sul PC. In questo manuale si vedrà quanto è facile aggiungere piccole porzioni di codice assembler in-line per ottenere un aumento di prestazioni notevole.

Vi è una serie di operazioni che semplicemente può essere eseguita con maggiore facilità utilizzando il linguaggio assembler. Ad esempio la scrittura di parti di un sistema operativo, utility di sistema, driver e così via. Molti di questi programmi otterranno anche grandi benefici prestazionali resi possibili dall'ottimizzazione per il Pentium.

Infine è utile imparare e utilizzare il linguaggio assembler poiché può essere un'occupazione divertente e una sfida. A maggior ragione questo si applica alla programmazione per il Pentium. Essendo la prima CPU della famiglia 80x86 dotata di più di una pipeline, presenta nuove e interessanti opportunità che erano assenti nelle CPU 80x86 precedenti.

Come procedere

I programmatori che non conoscono l'uso del linguaggio assembler (o comunque del linguaggio assembler sulle CPU della famiglia Intel 80x86) dovrebbero partire dai Capitoli 1 o 2. Chi conoscesse l'uso dei numeri binari ed esadecimali può passare direttamente al Capitolo 2.

I programmatori che già conoscono l'uso del linguaggio assembler 80x86 possono ignorare o semplicemente sfogliare i primi cinque capitoli.

I programmatori assembler più esperti possono partire dai Capitoli 8 o 9, in base al proprio livello di esperienza nella programmazione del Pentium. La maggior parte del materiale presentato nei capitoli più avanzati (Capitoli 16-19) si basa in misura notevole sulle informazioni presentate nei Capitoli 8-15

I capitoli sono suddivisi in parti:

Capitoli 1-3	*Introduzione e contesto storico*
Capitoli 4-7	*La famiglia 80x86*
Capitoli 8-12	*Introduzione al Pentium e ai relativi strumenti di sviluppo*
Capitoli 13-15	*Programmazione superscalare con il Pentium*
Capitoli 16-19	*Argomenti avanzati*
Capitolo 20	*Esempi di architetture superscalari*

Ringraziamenti

La realizzazione di questo manuale e particolarmente degli argomenti tecnici più avanzati non è stata un'impresa facile, e sarebbe stata impossibile senza l'aiuto di un gran numero di persone. In particolare sono state preziose le informazioni tecniche fornite da Frank van Gilluwe, Dave Horn, Rob Larson, Terje Mathisen e Stuart McCarley. Harlan Stockman (hwstock@sandia.gov) ha fornito la maggior parte del codice e dei test per il capitolo riguardante le operazioni in virgola mobile.

Parte prima

INTRODUZIONE E CONTESTO STORICO

Capitolo 1

Sistemi di numerazione

In questo capitolo si parlerà dei sistemi di numerazione binario, esadecimale e decimale. Chi avesse un'esperienza di lavoro con i numeri binari ed esadecimali, può passare direttamente al Capitolo 2.

Chi non ha mai provato a programmare con linguaggi di alto livello come il C, il BASIC o il Pascal, conoscerà probabilmente solo l'uso dei numeri decimali (numeri in base 10). Tutti noi siamo cresciuti utilizzando i numeri decimali, per contare il denaro, le ore, le varie unità di misura e perfino i canali della televisione. Tutto si basa sui numeri decimali tranne i circuiti interni dei computer e di altri dispositivi elettronici. I numeri decimali risultano così facili poiché noi siamo cresciuti insieme ad essi e, naturalmente, perché abbiamo 10 dita.

Ogni computer utilizza invece il sistema di numerazione binario. I numeri binari, o numeri in base 2, sono composti da due sole cifre, 0 e 1. I numeri decimali o numeri in base 10, usano dieci cifre, da 0 a 9. I computer utilizzano il sistema binario poiché i circuiti elettronici possono avere solo due stati, ovvero "attivo" o "non-attivo". Altri dispositivi possono impiegare proprietà fisiche differenti (ad esempio su un disco magnetico i dati binari possono essere memorizzati come aree magnetizzate o non magnetizzate oppure come aree magnetizzate verso nord o verso sud) ma l'effetto è lo stesso: i due stati delle cifre binarie.

Per conoscere meglio le cifre binarie, si partirà dai numeri decimali interi (ovvero i numeri 0, 1, 2, 3 e così via). I numeri decimali sono costituiti da sequenze di cifre decimali. Ogni cifra fa riferimento a due fattori che vengono moltiplicati fra loro. Il primo fattore è la cifra stessa (da 0 a 9). Il secondo fattore varia a seconda della posizione della cifra all'interno del numero intero. La cifra più a destra sarà moltiplicata per 1. La cifra successiva dovrà essere moltiplicata per 10, la successiva per 100 e così via. Spostando una cifra di una posizione, se ne moltiplica il valore per 10 e questo è esattamente il motivo per cui questo tipo di numerazione si dice "in base 10".

Ad esempio, il numero 3406 viene interpretato nel seguente modo:

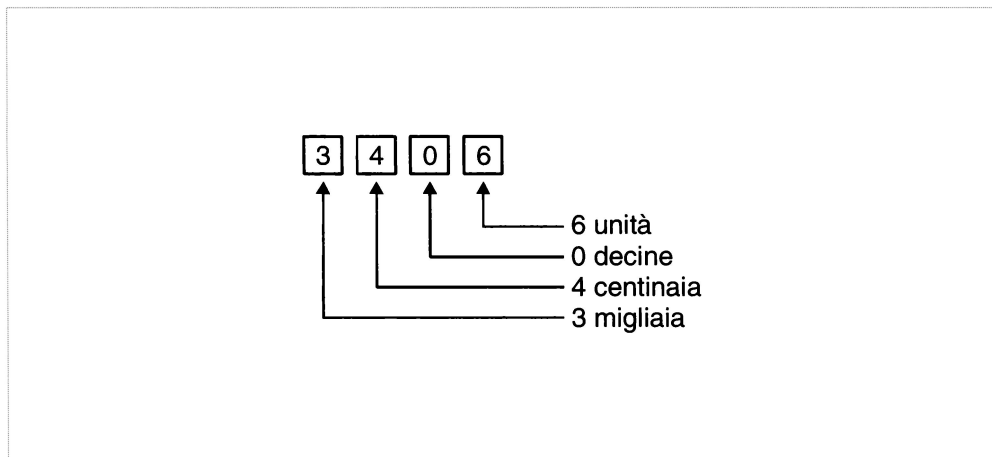



Figura 1.1 Esempio di numero decimale.

cifra	**x**	**moltiplicatore**	**=**	**totale**	**posizione**
6	x	1	=	6	0
+0	x	10	=	00	1
+4	x	100	=	400	2
+3	x	1000	=	3000	3
				3406	

Naturalmente in questo caso è ovvio che il valore di 3406 sia proprio 3406. Ma questo calcolo ha più senso se si deve eseguire una conversione di un numero da una base a un'altra. Questo stesso processo può essere eseguito per i numeri codificati in qualsiasi base. In particolare si deve notare che il moltiplicatore equivale alla base elevata a una potenza equivalente alla posizione. Nell'esempio precedente, il 4 viene moltiplicato per cento, ovvero per 10^2 (10 elevato al quadrato).

Ogni cifra di un numero binario è chiamata "bit", abbreviazione di "binary digit". Dunque quando si utilizza un numero binario, ad esempio composto da quattro bit, il valore di ciascun bit sarà progressivamente sempre più grande. Il primo bit (il bit 0) avrà un valore pari a 1, ovvero 2^0. Il bit successivo avrà un valore pari a 2, ovvero 2^1. Il bit successivo avrà un valore pari a 4, ovvero 2^2. L'ultimo bit avrà un valore pari a 8, ovvero 2^3. Ogni bit ha dunque un valore pari al doppio del valore del bit precedente. Analogamente, nella numerazione decimale, ogni cifra ha un valore pari a dieci volte il valore del numero della cifra che si trova alla sua destra.

Un byte è un numero binario che contiene 8 bit. Quando tutti i bit del byte sono uguali a 1 (il numero più grande inseribile in un byte), il valore (in base 10) del byte è 255. Dunque un byte può contenere un valore compreso fra 0 e 255.

Nell'esempio seguente si proverà a convertire un numero binario in un numero in decimale.

cifra binaria	x	moltiplicatore decimale	=	totale decimale	posizione
1	x	1	=	1	0
+0	x	2	=	0	1
+1	x	4	=	4	2
+1	x	8	=	8	3

La somma di numeri binari è molto semplice, poiché vi sono solo quattro possibili combinazioni di numeri da sommare. Al contrario, nel caso dei numeri in base 10 le combinazioni sono 100. Quello che segue è l'elenco di tutte le possibili combinazioni di bit.

bit 1	bit 2	risultato	
0	0	0	
0	1	1	
1	0	1	
1	1	0	con riporto (carry)

Per sommare due numeri binari, si segue la stessa procedura impiegata per i numeri decimali. Per questo esempio si proverà a sommare fra loro i numeri 001101001 e 00010001:

binario	decimale
01101001	105
00010001	17
01111010	122

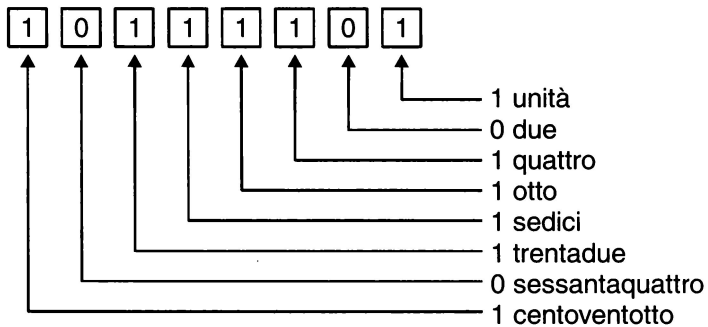



Figura 1.2 Esempio di numero in formato binario.

1.1 I numeri esadecimali

Il sistema esadecimale si basa su 16 cifre (ovvero è un sistema in base 16). Ma poiché per le cifre vi sono solo 10 simboli (da 0 a 9), questo pone un piccolo problema. Tale problema ha però varie soluzioni. Ad esempio si potrebbe utilizzare le prime 16 lettere dell'alfabeto. In alternativa si potrebbe utilizzare una serie di simboli interamente nuovi. Tuttavia si è preferito utilizzare una convenzione che prevede l'uso delle normali cifre decimali seguite dalle prime 6 lettere dell'alfabeto, ovvero le lettere da A a F che corrispondono ai valori da 10 a 15.

Ora si proverà a convertire un numero esadecimale prima in decimale e poi in binario. Il numero esadecimale è 3A21h.

cifra esadecimale	**x**	**moltiplicatore decimale**	**=**	**totale decimale**	**posizione**
1	x	1	=	1	0
+2	x	16	=	32	1
+A	x	256	=	2560	2
+3	x	4096	=	12288	3
				14881	

La conversione di un numero esadecimale in un numero binario è un processo completamente diverso e molto più facile. Ogni cifra esadecimale non è altro che una rappresentazione di quattro cifre binarie (4 bit). Questo è proprio il motivo per il quale il sistema di numerazione esadecimale è così utilizzato. Dunque si può pensare al sistema esadecimale come a una versione "abbreviata" del sistema binario. La Tabella 1.1 mostra le varie corrispondenze fra le cifre esadecimali e le rispettive cifre binarie. Dunque, per eseguire una conversione in cifre binarie, basterà applicare tale tabella in successione a ogni cifra esadecimale. Quindi la conversione del numero esadecimale 3A21h in cifre binarie può essere eseguita nel seguente modo:

```
numero esadecimale:  3     A     2     1
binario:             0011  1010  0010  0001
```

Per semplificare l'interpretazione dei numeri, negli esempi di questo libro verranno impiegati sia numeri esadecimali che numeri decimali. Per chiarezza, i primi saranno seguiti dalla lettera "h". Gli indirizzi di memoria sono sempre specificati in numeri esadecimali e si presentano nel formato "segmento:offset" (4 cifre esadecimali, seguite dal segno ":" e da altre 4 cifre esadecimali). Il significato del formato segmento:offset verrà discusso nei prossimi capitoli.

```
1234:5678    segmento 1234h, offset 5678h
123          numero decimale
123h         numero esadecimale
```

Nel Capitolo 3 si parlerà dell'uso di un "Assembler", ovvero un programma che si occupa di tradurre il codice scritto dall'utente in un formato che possa essere letto dalla macchina. Normalmente gli Assembler richiedono che i numeri esadecimali inizino con la cifra 0, poiché anche i nomi di variabili e le etichette dei programmi iniziano con caratteri alfabetici. Pertanto, se un numero esadecimale inizia con le lettere da A a F, il numero dovrà essere preceduto da uno 0.

```
0FFh    ; corrispondente esadecimale per il numero decimale 225
FFh     ; errore se si vuole indicare il valore decimale 255
        ; OK se si intende la variabile chiamata FFH
```

1.2 Numeri con segno

Come si è visto, un byte può contenere un numero compreso fra 0 e 255. Ma talvolta è necessario utilizzare anche numeri negativi.

Lo stesso byte che può contenere i numeri positivi da 0 a 255 può anche contenere i numeri con segno compresi fra -128 e 127. I valori da 128 a 255 vengono impiegati per i numeri negativi: dunque a 255 corrisponderà il numero -1, a 254 il numero -2 e così via. Può sembrare un metodo strano e confuso ma tutto si chiarirà meglio con un esempio. Si immagini di avere un'automobile con un contachilometri composto da cinque cifre e pertanto può visualizzare i numeri da 0 a 99.999 chilometri. Si supponga ora che l'automobile possa andare all'indietro e che anche il contachilometri proceda all'indietro.

Tabella 1.1 Equivalenze fra cifre decimali, esadecimali e binarie.

Decimali	Esadecimali	Binarie
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0100
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

Utilizzando un'automobile in cui il contachilometri riporta l'indicazione 00000, procedendo all'indietro per un chilometro, il contachilometri non riporterà l'indicazione -1 ma 99.999. Procedendo all'indietro per un altro chilometro, si leggerà l'indicazione 99.998. Ora, procedendo in avanti per 3 chilometri si leggerà l'indicazione 00001. Ovviamente, al numero 99.999 corrisponderà il valore -1, al numero 99.998 corrisponderà il valore -2 e così via.

Ora si proverà a reinterpretare il tutto in termini di numeri binari. Si immagini di installare un contachilometri formato da otto cifre binarie. Partendo da 00000000 e procedendo all'indietro per un chilometro si leggerà l'indicazione 11111111 (che in decimale corrisponde al numero 255). Procedendo all'indietro per un altro chilometro si leggerà l'indicazione 11111110 (equivalente al numero decimale 254). Procedimento in avanti per 3 chilometri si leggerà l'indicazione 00000001 (che equivale al numero decimale 1).

Rimane solo la difficoltà di determinare quando un numero ha il segno e quando invece non ha segno. In realtà questo è impossibile poiché entrambi i numeri hanno esattamente lo stesso aspetto. In alcuni linguaggi di alto livello è possibile dichiarare variabili di un determinato tipo (interi, interi con segno così via) per consentire al compilatore o all'interprete di determinare il tipo del numero e, di conseguenza, di gestire in modo corretto ogni variabile. In assembler tutto ciò deve essere eseguito dal programmatore. In particolare ai numeri con segno e senza segno corrispondono diverse istruzioni assembler, come si vedrà nei prossimi capitoli.

1.3 Overflow numerico

Anche solo avendo a che fare con numeri interi positivi, possono verificarsi dei problemi. Si supponga di avere una variabile costituita da un byte (per la quale sono ovviamente consentiti valori compresi fra 0 e 255), che tale variabile contenga il valore 255 e che le venga sommato il numero 1 per ottenere 256. Così facendo si commette un errore poiché un byte può contenere al massimo il valore 255, quindi è necessario tenere in considerazione questa condizione. Sarebbe un po' come avere un'automobile con 99.999 chilometri e al chilometro successivo cercare di convincere qualcuno che l'automobile è appena uscita dalla fabbrica. Basta dare un'occhiata alla carrozzeria per rendersi conto che anche se il contachilometri mostra l'indicazione 00000, l'automobile ha in realtà percorso 100.000 chilometri. In particolare si noti che il valore 100.000 corrisponde al massimo numero visualizzabile dallo strumento più 1. Ma si torni al problema di sommare 1 unità al numero 255. Poiché il computer utilizzerà solo 8 cifre binarie (255) si otterrà un risultato pari a 0 e, poiché i byte non hanno la carrozzeria, occorre prevedere qualche indicazione che segnali il superamento dei limiti del byte (condizione di overflow). Per questo motivo, i computer hanno una serie di flag che si attivano nel momento in cui si verifica una determinata condizione, come ad esempio un overflow numerico. Il flag di riporto "carry" si attiva quando il risultato della somma di due byte è maggiore di 255. Ma questo argomento verrà affrontato in seguito.

Ora si immagini di lavorare con byte dotati di segno (numeri compresi fra -128 e 127). Sommando al numero 125 il numero 10 si ottiene il numero 135. Ma quando

vengono utilizzati numeri con segno, il numero 135 corrisponde al numero -121. In questo caso sembra che basti verificare se il risultato doveva essere negativo per determinare se si è verificato un overflow. Errore: in realtà, se si sommano due interi positivi con segno, un risultato negativo indicherebbe un overflow. Ma cosa accadrebbe se i due numeri sommati fossero negativi? Ci si attenderebbe un risultato negativo. E quando i due numeri sono uno negativo e l'altro positivo? A tale scopo il computer ha un "flag di overflow" che segnala proprio questa condizione. Quando si sommano due numeri positivi con segno o due numeri negativi con segno, se il risultato ha un segno diverso dai due addendi, viene attivato il flag di overflow. Quando si sommano due numeri di segno diverso, non vi può essere overflow. Analogamente, il flag di overflow si attiva quando la sottrazione di due numeri con segno diverso causa un overflow. Questi argomenti verranno discussi più approfonditamente nel Capitolo 4.

1.4 Dimensioni dei dati

A seconda dello scopo per il quale viene utilizzata una variabile, si può decidere che essa possa contenere una gamma più o meno estesa di valori. In questa sezione ci si occuperà dei vari tipi di dati disponibili sul microprocessore 8088. Questo argomento verrà trattato in modo più dettagliato nei prossimi capitoli.

Il byte è la più piccola unità di memorizzazione indirizzabile nella memoria di un computer. Ogni byte ha un proprio indirizzo e può essere singolarmente letto o scritto dalla CPU. Per leggere un solo bit è perciò necessario leggere in un registro uno o più byte (i registri verranno descritti nel Capitolo 4). Se si deve scrivere un solo bit, si deve prima leggere un byte, modificare il bit e quindi riscrivere in memoria l'intero byte. Vi è anche un concetto intermedio fra bit e byte, ovvero il nibble, composto da 4 bit. Alcuni dati vengono infatti memorizzati uno per nibble ma dovranno comunque essere letti e scritti un byte alla volta.

I byte possono rappresentare:

Interi	Interi con o senza segno composti da 8 bit.
Caratteri	Normalmente i caratteri ASCII.

Una word è composta da 16 bit ovvero due byte. Anche le word, come i byte, possono essere liberamente lette o scritte nella memoria. Le word possono essere utilizzate per:

Interi	Interi con o senza segno a 16 bit.
Segmenti	Descrittori di segmenti a 16 bit sia in modalità reale che in modalità protetta.
Offset	Valori di offset a 16 bit rispetto all'inizio del segmento.

Una dword è composta da 32 bit, ovvero due word o quattro byte. Nel formato dword vengono comunemente memorizzati quattro tipi di dati:

Interi	Interi con o senza segno a 32 bit.
Puntatori far	Combinazioni di segmenti e offset.
Valori float	Valori in virgola mobile in precisione semplice.
Offset	Offset a 32 bit per i segmenti estesi in modalità protetta.

Nella parte di dichiarazione dei dati di un programma, DB dichiara un byte, DW dichiara una word e DD dichiara una dword.

1.5 Tipi di memorizzazione

Talvolta non è chiaro il formato in cui vengono memorizzati i valori più grandi di 8 bit. Sui computer della famiglia 80x86 (8088/86, 80286 e così via), una word viene normalmente memorizzata in modo che il byte meno significativo si trovi nell'indirizzo di memoria più basso e il byte più significativo si trovi nell'indirizzo di memoria più alto. Altre CPU memorizzano i valori multi-byte in senso opposto, in particolare questo avviene nel caso dei microprocessori della famiglia Motorola 680x0. In alcuni casi si dice che il metodo impiegato dai microprocessori 80x86 è un metodo "invertito".

Il formato dei byte Intel conserva i valori a partire dal byte meno significativo. L'altro formato memorizza i dati a partire dal byte più significativo.

Il metodo impiegato da Intel può in alcuni casi causare problemi. Tuttavia si deve riconoscere che vi sono buoni motivi per impiegare questo metodo. Tutti noi siamo abituati a leggere le parole partendo da sinistra e procedendo verso destra. Si tratta però di una convenzione assolutamente arbitraria. Questo è lo stesso motivo per il quale tendiamo a scrivere i numeri in ordine crescente procedendo da sinistra verso destra. Osservando una mappa stradale, gli indirizzi potrebbero avere il seguente aspetto (in questo esempio viene ignorata la convenzione che assegna i numeri pari a un lato della strada e i numeri dispari all'altro lato).

Prima strada	4	3	2	1
Seconda strada	4	3	2	1

In pratica ogni casa ha un indirizzo corrispondente a una singola cifra e tali numeri crescono procedendo verso ovest. Ora, si immagini di dover dipingere su ogni strada un grosso numero decimale composto da quattro cifre, leggibili da un elicottero. Ogni cifra occuperà lo spazio di fronte a una delle case della via. Sulla prima strada si scriverà il numero 8022 e sulla seconda strada si scriverà il numero 3077.

Prima strada	4	3	2	1	← indirizzo
	8	0	2	2	← numero dipinto
Seconda strada	4	3	2	1	← indirizzo
	3	0	7	7	← numero dipinto

Considerando il nord in direzione del margine superiore della pagina, se l'elicottero vola da sud a nord, i numeri appariranno orientati correttamente ma se l'elicottero vola da nord, est o ovest, sarà molto più difficile leggere i numeri. Quello che si intende dimostrare è che il modo in cui si sceglie di rappresentare le cose su carta potrebbe non avere molto a che fare con il loro utilizzo pratico.

Nel caso dei computer, il metodo impiegato da Intel ha un preciso utilizzo pratico. Tale utilizzo ha a che fare con una delle più importanti funzioni svolte da un compu-

ter: la somma. Si provi a sommare i numeri sulla prima e sulla seconda strada e a inserire il risultato nella terza strada. A tale scopo si dovranno sommare i numeri corrispondenti alle posizioni 1 della prima e della seconda strada per inserire il risultato nella posizione 1 della terza strada e così via.

Prima strada		4	3	2	1	←	indirizzo
		8	0	2	2	←	valore
Seconda strada		4	3	2	1	←	indirizzo
		3	0	7	7	←	valore
Terza strada	5	4	3	2	1	←	indirizzo
	1	1	0	9	9	←	valore

Come si noterà, nella posizione 4 della terza strada si deve risolvere un piccolo problema. Il risultato è composto da due cifre e quindi è necessario aggiungere un altro indirizzo alla strada. Il metodo precedente corrisponde a una somma svolta utilizzando il sistema di memorizzazione dei byte impiegato da Intel (lo stesso metodo può essere utilizzato anche per la sottrazione).

Utilizzando la metafora degli indirizzi di una strada si è in realtà parlato del modo in cui i dati vengono conservati nella memoria del computer. Sostituendo alle cifre degli esempi precedenti un byte di dati, si ottiene un modello del modo in cui i dati vengono conservati in memoria.

Ecco come viene eseguita la stessa somma utilizzando il metodo impiegato da Motorola:

Prima strada		1	2	3	4	←	indirizzo
		8	0	2	2	←	valore
Seconda strada		1	2	3	4	←	indirizzo
		3	0	7	7	←	valore
Terza strada	0	1	2	3	4	←	indirizzo
	1	1	0	9	9	←	valore

Come si noterà, nella terza strada si è dovuto aggiungere un nuovo indirizzo uguale a zero. In questo caso non vi è alcun problema ma gli indirizzi non possono essere negativi. Se si utilizza il sistema Motorola, occorre assicurarsi, prima di eseguire l'operazione, che il numero di indirizzi disponibili sia sufficiente per l'operazione mentre nel caso della tecnica impiegata da Intel, l'indirizzo potrà essere aggiunto alla fine, in modo molto più semplice. Come si è imparato alle elementari, le somme o le sottrazioni devono essere eseguite partendo dalla cifra meno significativa. Come si può notare, il metodo impiegato da Motorola richiede che si parta dall'indirizzo più alto per procedere verso il basso.

Capitolo 2

Che cos'è il linguaggio assembler

Questo manuale tratta l'argomento della programmazione in linguaggio assembler per aiutare a sviluppare con rapidità programmi più efficienti per la famiglia di microprocessori Intel 80x86.

L'obiettivo è quello di aiutare il lettore a diventare un buon programmatore assembler in grado di produrre in modo efficiente programmi e/o subroutine ottimizzati. Pertanto si partirà dalle basi, rappresentate dai microprocessori 8088 e 8086, per poi procedere progressivamente verso il Pentium.

I programmatori assembler più esperti potranno invece saltare i primi quattro capitoli. Sarà sufficiente leggere le prime due pagine per capire se un capitolo può essere saltato, sfogliato o letto con attenzione.

2.1 Che cos'è il linguaggio assembler

Il linguaggio assembler è un linguaggio specifico di un computer o di una famiglia di computer. Uno degli obiettivi della maggior parte degli linguaggi di alto livello, come il C, il BASIC, il Pascal o il FORTRAN è quello di consentire ai programmatori di scrivere istruzioni che abbiano un aspetto più simile al modo in cui verrebbe descritto il problema in linguaggio naturale. Naturalmente, come si sa, questo è un obiettivo non ancora raggiunto.

Ad esempio, in BASIC si potrebbe scrivere la seguente istruzione:

```
IF Conto < 0 THEN PRINT "Conto in rosso"
```

Mentre l'aspetto del linguaggio assembler è più complesso e prolisso:

```
CMP   Conto, 0
JA    OK
MOV   DX,    OFFSET Messaggio
MOV   AH,    9
INT   21h
```

Per il momento non ci si deve preoccupare di comprendere il significato delle istruzioni assembler poiché se ne parlerà in seguito. Interessa invece dimostrare che in assembler si deve dire al computer ogni singola operazione che deve essere eseguita. Si può dire che la differenza principale fra il linguaggio assembler e un linguaggio di alto livello corrisponde alla differenza che esiste fra chiedere a qualcuno di calcolare il consumo della propria auto oppure dover indicare alla stessa persona ogni singolo tasto della calcolatrice che deve essere premuto.

Dunque, dal momento che è necessario eseguire un lavoro così dettagliato, perché mai si dovrebbe programmare in linguaggio assembler? I motivi sono vari. Innanzitutto i computer sono in grado di comprendere direttamente solo il linguaggio macchina e un programma assembler può, in genere, essere direttamente tradotto in linguaggio macchina. Tutti gli altri linguaggi devono invece essere sottoposti a una fase di compilazione (per la traduzione dei programmi in linguaggio assembler) oppure i programmi devono essere eseguiti da un interprete. Dunque vi deve essere qualcuno che scriva i programmi in assembler per ogni diverso tipo di computer. Questo "qualcuno" può essere un programmatore assembler oppure può essere un compilatore (come nel caso del linguaggio C). Ma chi scrive il compilatore, dovrà conoscere il funzionamento del linguaggio assembler in modo da poter tradurre in modo corretto le istruzioni del linguaggio di alto livello nelle corrispondenti istruzioni assembler.

Ma vi sono molti altri motivi che spingono all'uso del linguaggio assembler. Grazie al linguaggio assembler è possibile scrivere programmi più veloci e compatti rispetto a quelli producibili dalla maggior parte dei compilatori o interpreti. Talvolta la differenza in termini di dimensioni o velocità è sorprendente. In assembler è possibile accedere a ogni singolo aspetto o funzionalità del microprocessore. Ad esempio, è possibile scrivere programmi che manipolano lo stack di sistema, che comunicano con dispositivi come il modem o le casse acustiche e che utilizzano le istruzioni matematiche in virgola mobile. Anche se molte di queste operazioni possono essere eseguite anche nei vari linguaggi di alto livello, raramente ciò viene eseguito in modo efficiente senza richiamare una funzione o una subroutine scritta in linguaggio assembler.

Ma non sempre la programmazione in linguaggio assembler raggiunge un livello così basso e dettagliato. Così come avviene in ogni altro linguaggio, i progetti di una certa dimensione possono essere suddivisi in più componenti. Le operazioni più comuni possono essere svolte da procedure di basso livello. Inoltre è possibile realizzare o acquistare librerie simili a quelle del C. In pratica, il linguaggio assembler consente di ridurre significativamente il tempo di esecuzione dei programmi. Questo è anche l'obiettivo principale di questo manuale. In particolare verranno studiati metodi utili per ottimizzare le subroutine o le funzioni comuni a molti programmi. Si possono utilizzare queste tecniche e questi esempi in progetti realizzati interamente in linguaggio assembler oppure si può utilizzare il linguaggio assembler per ottimizzare programmi scritti in C, Pascal, FORTRAN o BASIC.

2.2 Programmi più veloci e compatti

Ma perché i programmi assembler sono più veloci e compatti? I motivi sono vari. Innanzitutto, quando si scrive in assembler, si utilizza il minor numero di istruzioni

necessarie per eseguire una determinata operazione. In secondo luogo, non è necessario selezionare le istruzioni sulla base di modelli o regole predefinite (come avviene nella maggior parte dei compilatori). Si può cioè utilizzare il miglior compilatore disponibile, il proprio cervello. Inoltre è possibile selezionare l'istruzione migliore per ogni situazione. A seconda dei casi, l'istruzione migliore può essere la più veloce o la più compatta o entrambe le cose insieme. Questo significa che è sempre il programmatore a poter decidere quale sia la "migliore" istruzione a seconda dei casi.

Dunque vi sono vantaggi e svantaggi. In compilatori non possono (almeno per il momento) fare meglio degli uomini, ma generano codice molto più velocemente. Quindi si deve decidere se la velocità e/o la compattezza del codice è più importante rispetto al rapido sviluppo o alla possibilità di ricompilazione rapida per un'altra macchina. In generale, i compilatori hanno solo poche opzioni per controllare il modo in cui viene generato il codice. Ad esempio, nel C Microsoft, si può decidere di ottimizzare la velocità o le dimensioni. Oltre a questo è possibile attivare o disattivare determinate tecniche di ottimizzazione specifiche. L'unico modo in cui è possibile ottimizzare in modo selettivo parti di un programma per ottenere codice compatto in alcuni casi e veloce in altri casi consiste nel compilare il programma utilizzando file (o moduli) distinti.

2.3 Strumenti e terminologia

Nella parte rimanente di questo capitolo si parlerà della terminologia e degli strumenti che si utilizzano nello sviluppo del software. In ogni caso si tratta semplicemente di una breve introduzione poiché un esame approfondito degli strumenti software richiederebbe da solo un intero libro.

Che cosa sono i compilatori, gli interpreti e gli assembler

I compilatori, gli interpreti e gli assembler rientrano in una classe di strumenti nota con il nome generico di "traduttori". Un traduttore umano può ascoltare ciò che una persona dice in inglese e ripetere ciò che ha udito parlando in italiano. L'idea è che il traduttore operi come un intermediario. Anche se le due persone che cercano di comunicare possono conoscere in modo più o meno approfondito l'altra lingua, non conoscono abbastanza per comunicare in modo efficiente senza il traduttore. Nel caso dei computer, si può conoscere qualche aspetto del linguaggio macchina (vedere di seguito), ma certamente la macchina non conosce nulla del linguaggio umano. Il programma traduttore esegue proprio questo servizio. Si scrive un programma in un linguaggio comprensibile al programmatore e il traduttore lo converte in un linguaggio comprensibile da parte della macchina.

Linguaggio macchina

Il linguaggio macchina è il linguaggio che un computer è direttamente in grado di comprendere. Si tratta di un linguaggio molto difficile da capire ma è anche l'unico direttamente utilizzabile dai computer. Tutti i programmi e i linguaggi di programma-

zione generano o eseguono programmi in linguaggio macchina. Il linguaggio macchina è composto da istruzioni e dati formati unicamente da numeri binari. Il linguaggio macchina viene normalmente visualizzato in formato esadecimale per semplificarne leggermente la lettura. Il linguaggio assembler è molto simile al linguaggio macchina, tranne per il fatto che le istruzioni, le variabili e gli indirizzi hanno un nome.

Assembler (programmi assemblatori)

Un assembler è un programma di traduzione che accetta un file in codice assembler e lo converte in codice oggetto. Il codice oggetto è in pratica la stessa cosa del linguaggio macchina ma è strutturato in unità logiche in modo che sia possibile posizionarlo a piacere in memoria e unirlo ad altri file di codice oggetto. Nella maggior parte dei casi un'istruzione assembler viene tradotta direttamente in un'istruzione in linguaggio macchina. Per gli esempi di questo libro verranno impiegati gli assembler Microsoft MASM e Borland TASM.

Compilatori

Un compilatore è un programma di traduzione che accetta un file scritto con un linguaggio di programmazione di alto livello (ad esempio il C, il BASIC o il Pascal) e lo converte in codice oggetto. In sostanza, un compilatore prende ogni istruzione scritta in codice sorgente e genera l'equivalente in linguaggio macchina. Una singola istruzione scritta in linguaggio di alto livello può generare da una a qualche decina di istruzioni in linguaggio macchina. Per gli esempi di questo libro verranno impiegati i compilatori Microsoft C/C++ e Borland C/C++.

Editor

Gli editor sono programmi che consentono di creare e modificare i file di testo. Infatti, i file di codice sorgente dei programmi sono a tutti gli effetti file di testo. In ambiente MS-DOS sono disponibili numerosi editor fra i quali il programma EDIT fornito a partire dal DOS 5.0. Per creare file di testo è anche possibile utilizzare programmi di videoscrittura ma in questo caso è necessario assicurarsi di salvare i file come semplici file di testo.

Moduli e librerie di codice oggetto

Un modulo oggetto è il file creato compilando o assemblando un singolo file sorgente di un programma. Questi file sono chiamati file di codice oggetto. Se si preferisce, i moduli di codice oggetto possono essere riuniti all'interno di librerie. La maggior parte dei linguaggi di alto livello è dotata di una o più librerie predefinite di funzioni che possono essere utilizzate per eseguire le varie funzioni messe a disposizione dal compilatore. Ad esempio, le librerie run-time del C contengono le funzioni printf, scanf e strcpy.

Vi è un'altra e importante differenza fra i concetti di "codice oggetto" e tutti gli altri tipi di "oggetti" di cui si parla nel campo dei computer. I termini "codice oggetto", "file .OBJ" e così via, fanno riferimento a file in linguaggio macchina che hanno un formato tale da consentirne il linking o l'esecuzione da parte del computer. Al contrario, con programmazione orientata agli oggetti si fa riferimento alle tecniche e alle funzionalità che consentono a un programmatore di scrivere funzioni facilmente riutilizzabili e/o estendibili da parte di altri programmatori. Alla tecnica di program-

mazione orientata agli oggetti sono collegate le parole ereditarietà, polimorfismo e incapsulazione. Questi concetti non hanno nulla a fare con il codice oggetto.

Linker

Un linker è un programma che unisce uno o più file di codice oggetto e (opzionalmente) i moduli di codice oggetto contenuti in una libreria, in modo da produrre un file eseguibile. Nel caso del DOS, tali file hanno estensione .EXE o .COM. Nel caso di Windows non esiste l'estensione .COM. Windows supporta un particolare formato eseguibile chiamato DLL (da Dynamic Link Library). In questo libro verranno impiegati il linker LINK Microsoft e TLINK Borland.

Interpreti

Gli interpreti sono programmi che leggono un programma scritto con un linguaggio di alto livello, determinano le azioni che devono essere eseguite e ne eseguono le istruzioni. Alcuni interpreti traducono i programmi in un formato intermedio (ma non in linguaggio macchina) e quindi eseguono il file tradotto in questo formato intermedio. Questo particolare formato viene chiamato semi-compilato o p-code. In particolare è interpretata la maggior parte delle versioni di BASIC. Sono interpretati anche linguaggi particolari come il PostScript e il REXX.

Disassembler

Un disassembler è un programma che legge un programma in linguaggio macchina e cerca di ricostruire il codice assembler che lo ha prodotto. Questa operazione è particolarmente difficile poiché non vi è alcuna differenza fra il codice e i dati: si tratta di semplici successioni di byte.

I disassembler più sofisticati, come ad esempio Sourcerer della V Communications, eseguono una complessa analisi e simulazione del programma in modo da separare correttamente il codice dai dati. L'impiego di un disassembler è molto utile per comprendere il modo in cui funzionano altri programmi e per apprendere trucchi e tecniche utili.

Il disassemblaggio di un programma non è sempre legale ma molto dipende da ciò che si fa con il codice risultante. In generale, non vi sono grossi problemi se si utilizza il codice di un prodotto di cui si è in possesso per proprio utilizzo personale o per meglio comprendere il funzionamento di un programma. Se invece si include in un proprio programma il codice tratto da un programma coperto da diritti d'autore e si distribuisce tale codice anche gratuitamente si violano i diritti esistenti sul programma. In caso di dubbi si consulti un legale esperto in tutela del software.

Debugger

I debugger sono particolari programmi che consentono di seguire il funzionamento dei programmi alla scoperta di errori o bug. In particolare i debugger consentono di interrompere l'esecuzione del programma in un determinato punto (chiamato breakpoint) per esaminare o modificare i valori delle variabili e dei registri. Il nome del debugger DOS è DEBUG. I debugger CodeView Microsoft e Turbo Debugger Borland sono molto più sofisticati e consentono di osservare il codice sorgente durante l'esecuzione del programma. Altri debugger molto sofisticati, come ad esempio il Soft-ICE Nu-Mega utilizzano particolari funzionalità dei microprocessori 80386 e suc-

cessivi per definire breakpoint hardware. Nel disco fornito è presente il debugger DEBUG32, un potente debugger compatibile DPMI a 32 bit per l'esecuzione in modalità protetta. Per una descrizione di questo debugger si consulti il Capitolo 6.

Emulatori di circuiti

Un emulatore di circuiti (ICE da In Circuit Emulator) è un dispositivo hardware che si inserisce nello zoccolo della CPU del computer. Tale emulatore segue i vari eventi hardware che si verificano nel sistema ed emula la CPU utilizzando, oltre la CPU, altri dispositivi hardware aggiuntivi. Il vantaggio derivante dall'uso di un ICE è legato alla possibilità di seguire ciò che avviene nelle varie parti della memoria di sistema, di creare breakpoint complessi e di raccogliere informazioni su ciò che avviene nella macchina durante l'esecuzione di un programma. Tutto ciò può essere eseguito praticamente a piena velocità. Lo svantaggio principale di questi strumenti è legato al loro prezzo. Utilizzando esclusivamente prodotti software si può ottenere la maggior parte delle funzionalità di un ICE utilizzando un prodotto chiamato Periscope Model IV.

Capitolo 3

Storia e architettura della famiglia di microprocessori 8086

Non è certo che la frase "Chi non conosce il passato è condannato a riviverlo" si applichi anche al mondo dei microprocessori, ma la storia della famiglia 80x86 dice molto sull'aspetto dei prodotti Intel attuali e futuri. Questo capitolo presenta una breve storia della famiglia di microprocessori 80x86 Intel e dei principali sistemi operativi che hanno impiegato tali microprocessori.

La storia della famiglia di microprocessori 80x86 inizia nel 1972 quando Intel Corporation iniziò a vendere un microprocessore a 8 bit chiamato 8008. L'obiettivo principale di questo microprocessore era quello di fungere da chip di controllo per lo schermo dei computer.

Quando i progettisti di computer parlano di microprocessori con struttura a 8 bit fanno riferimento, in generale, alle dimensioni degli indirizzi e dei percorsi seguiti dai dati. Ma molti utilizzano questo termine per far riferimento unicamente alle dimensioni dei registri generali interni, che influenzano direttamente le caratteristiche del software scritto per il chip. Anche il numero di linee per gli indirizzi è molto importante per lo sviluppo del software, poiché tale numero è direttamente proporzionale al numero di byte di memoria indirizzabile. Tuttavia, sono esistiti molti microprocessori a 32 bit con capacità di memoria virtuale i quali avevamo meno di 32 linee di indirizzamento.

Nel 1973 Intel annuncia il microprocessore 8080, un altro chip a 8 bit. Tale chip funzionava alla velocità di 2 Mhz (1 Mhz corrisponde a un milione di cicli al secondo). Il chip 8080 venne impiegato in molti dei primi computer, incluso l'IMSAI, l'Altair ed altri verso la metà degli anni settanta. L'8080 aveva un bus dati a 8 bit e un bus indirizzi a 16 bit che gli consentiva di indirizzare un totale di 64 KB di memoria.

Naturalmente, perché questi primi microcomputer potessero rendersi utili era necessario realizzare il relativo software. Nella metà degli anni settanta, Digital Resource Inc. (DRI) iniziò la vendita del sistema operativo CP/M. Si trattava del primo sistema operativo per microcomputer commerciali progettato per essere utilizzato su macchine costruite da varie società. La sigla CP/M significa Control Program for Microcomputers (programma di controllo per microcomputer). Alla fine degli anni '80 DRI ha prodotto il sistema operativo DR-DOS, simile al sistema MS-DOS. Ora DRI fa parte di Novell.

Uno dei progettisti del microprocessore 8080 lasciò l'Intel per creare la Zilog e iniziare la progettazione dello Z-80, un altro chip a 8 bit. Lo Z-80 era di gran lunga superiore all'8080 e funzionava a una velocità doppia, 4 Mhz. Nel 1976 Intel inizia la produzione dell'8085 che però non ebbe mai il successo dello Z-80 anche se è ancora oggi in vendita soprattutto per essere impiegato all'interno di moduli di calcolo. La maggior parte dei programmi per lo Z-80 è stata realizzata per il sistema operativo CP/M; tali programmi erano comunque scritti per essere compatibili con il microprocessore 8080. Grazie all'alto livello di compatibilità, tali programmi utilizzavano lo Z-80 come se fosse un veloce 8080, sfruttando talvolta le nuove funzionalità introdotte dallo Z-80. I programmatori e i clienti cercavano e spesso richiedevano la compatibilità con tutti i computer 8080 e Z-80 precedentemente venduti. I successi e i fallimenti dello Z-80 sono una grande lezione. Acquisì rapidamente fette di mercato grazie alla sua velocità e compatibilità. Ma fallì la realizzazione di applicazioni specifiche per lo Z-80, lasciando all'8088 il posto di minimo comune denominatore.

Nel 1978 Intel annunciò il microprocessore 8086. Si trattava di un chip a 16 bit con un bus dati a 16 bit e un bus indirizzi a 20 bit. Per semplificare la transizione verso il nuovo chip, l'8086 era in grado di fare qualsiasi cosa facesse l'8080 anche se non era compatibile a livello di codice sorgente e codice eseguibile con l'8080. Tuttavia, i programmi scritti per l'8080 potevano essere facilmente tradotti nel linguaggio assembler dell'8086 in quanto i due microprocessori avevano istruzioni uguali o simili. Ma in generale dopo la traduzione i programmi erano comunque in grado di indirizzare solo 64 KB di memoria anche se l'8086 poteva indirizzare 1 MB (2^{20} = 1.048.576 byte). Intel produsse perfino un programma che eseguiva la traduzione automatica del linguaggio assembler 8080 nel linguaggio assembler 8086. Nel 1979 molte società producevano schede add-on e computer con chip 8086. L'unico problema era la scarsa disponibilità di programmi per l'8086.

Fu allora che Tim Patterson scrisse un sistema operativo simil-CP/M per l'8086. Questo prodotto era noto come il nome di 86-DOS e venne venduto con i sistemi della Seattle Computer Products. Microsoft acquistò l'86-DOS che poi divenne l'MS-DOS per il PC IBM e il resto è storia recente. Ma non fu tutto così semplice. Nel 1981 IBM annunciò il PC. Contemporaneamente, IBM annunciò la disponibilità di tre sistemi operativi: PC-DOS (la versione IBM di MS-DOS), CP/M-86 (prodotto da DRI) e il sistema UCSD p-system.

Questi tre sistemi operativi cercavano di essere compatibili con i sistemi precedenti. In particolare il sistema UCSD p-system era compatibile a livello di codice sorgente con molti sistemi operativi. Ma trattandosi di un sistema operante con codice p-code (compilato in token), era molto più lento dei suoi concorrenti (la compilazione in token è completamente diversa dalla compilazione in codice oggetto in quanto viene prodotto un codice intermedio che richiede l'impiego di un interprete; questo è il

motivo per il quale i linguaggi compilati sono normalmente molto più veloci dei linguaggi interpretati). Sia il CP/M-86 che l'MS-DOS (PC-DOS) avevano un alto livello di compatibilità con il CP/M. L'unico vero svantaggio per l'MS-DOS era l'impiego di un formato diverso per i dischi. Questo significava che i computer 8080/Z-80 con programmi CP/M erano in grado di scambiare con facilità i dati con i computer 8086 con sistema operativo CP/M-86. Questo fattore fu determinante nel primo anno di vita del PC IBM. Ma trascorso il primo anno, la base di PC IBM installati era così estesa che non era più richiesta la compatibilità con il più vecchio sistema CP/M. Il fatto che il sistema operativo MS-DOS impiegasse un nuovo sistema di formattazione dei file si rivelò allora un vantaggio grazie alla sua notevole velocità.

Il PC IBM impiegava la CPU 8088 e non l'8086. L'8088, introdotto nel 1979 aveva la stessa struttura interna della dell'8086 ma impiegava solo 8 bit per il trasferimento dei dati. Il software era comunque in grado di leggere e scrivere 16 bit di dati per volta, ma l'hardware di interfacciamento con la memoria suddivideva l'accesso in due operazioni a 8 bit. In questo modo i progettisti hardware potevano continuare a utilizzare componenti a 8 bit, più conosciute ed economiche. Probabilmente IBM ha scelto di utilizzare l'8088 per abbassare il più possibile il prezzo dei sistemi.

Alcuni sono convinti che fu un errore da parte dell'IBM la scelta dell'8088 al posto dell'8086. Ma questo non ha molto senso poiché il software per l'8088 era esattamente lo stesso utilizzato dall'8086. Entrambi hanno registri a 16 bit e linee di indirizzamento a 20 bit. Semplicemente l'8086 aveva un bus dati a 16 bit mentre l'8088 aveva un bus dati a 8 bit. È opinione dell'autore che non vi sia alcun motivo per chiamare l'8088 una macchina a 8 bit. L'unica differenza è nel progetto hardware che d'altra parte è notevolmente cambiato nell'ultimo decennio. Ad esempio, il PC-AT IBM aveva un nuovo progetto di bus così come la serie di macchine PS/2 che impiegava l'architettura Microchannel (MCI). Le macchine EISA hanno ancora un altro tipo di bus molto più veloce che però accetta le schede progettate per il bus PC-AT. Le nuove macchine impiegano la tecnologia Local Bus (VESA o PCI) ma possono utilizzare anche schede del vecchio tipo.

3.1 La lezione della compatibilità

Fino ad ora tutto sembra molto chiaro: i clienti chiedevano una piena compatibilità con tutto il software precedente. Preferibilmente era richiesta una compatibilità binaria che consentiva di utilizzare direttamente i programmi senza necessità di riassemblaggio o ricompilazione. Inoltre era richiesta una compatibilità nel formato dei dati anche se molti non lo consideravano un fattore importante. Si era disposti a sacrificare una parte di compatibilità in cambio di un aumento di prestazioni.

3.2 I coprocessori matematici

Uno dei migliori argomenti di marketing del PC IBM originale era la presenza di uno zoccolo vuoto per il coprocessore matematico 8087. Anche se una piccolissima per-

centuale di clienti ha poi eseguito l'aggiornamento del proprio sistema per includere questa opzione, il fatto che fosse presente lo zoccolo per il coprocessore rendeva il sistema molto più appetibile.

Il coprocessore matematico 8087 è un dispositivo indipendente in grado di eseguire operazioni in virgola mobile molto più velocemente rispetto al microprocessore principale che eseguiva un'emulazione con numeri interi. Poiché l'8087 è connesso direttamente alla CPU 8088/8086, è in grado di decodificare le istruzioni in modo sincrono rispetto alla CPU. Il coprocessore matematico, chiamato anche FPU, riconosce ed esegue solo le istruzioni specificatamente destinate ad esso. Mentre il coprocessore esegue un'istruzione, segnala alla CPU il fatto che è impegnato (BUSY) da un'elaborazione. L'istruzione WAIT fa in modo che la CPU si fermi sino al termine del segnale BUSY. Se si verifica un'eccezione in virgola mobile, il coprocessore può interrompere la CPU. L'8087 esegue operazioni aritmetiche a 80 bit e operazioni di confronto su vari formati di dati. I dati in virgola mobile utilizzati dal coprocessore usano il formato IEEE per numeri in virgola mobile con precisione semplice e doppia.

Nel 1982, Intel ha annunciato i microprocessori 80186 e 80188. Queste CPU non vennero però quasi per nulla impiegate nei computer desktop. Principalmente venivano impiegate per il controllo di altri sistemi elettronici. L'80188 è uguale all'80186 tranne per il bus dati a 8 bit (come l'8088). L'80188 e l'80186 avevano alcune funzionalità aggiuntive normalmente utilizzate nei sistemi elettronici, come ad esempio 3 timer interni, un controller per interrupt, un controller DMA e un generatore clock. Inoltre l'80188 e l'80186 aggiungevano 10 nuove istruzioni alla CPU e miglioravano la velocità della maggior parte delle istruzioni.

3.3 L'80286

Nel 1982, Intel annunciò l'80286. Il 286 venne impiegato per la prima volta da IBM nel 1984 quando annunciò il PC AT a 6 Mhz. La sigla AT stava per Advanced Technology. Il 286 era in grado di eseguire tutto il software precedente ma aveva una modalità di programmazione aggiuntiva: la modalità protetta. In modalità protetta, ogni segmento di programma non viene indirizzato con un indirizzo fisico ma tramite un selettore di segmento. Questo consentì di aumentare la massima memoria indirizzabile da 1 MB a 16 MB.

La modalità protetta è particolarmente utile nei sistemi operativi multiutente in quanto "protegge" il codice e i dati di un programma da possibilità di lettura o scrittura da parte di un altro programma. Inoltre la modalità protetta tende a impedire che un programma possa accidentalmente auto-danneggiarsi.

Ma d'altra parte la modalità protetta impedisce (o rende molto difficile) la possibilità che un programma esegua quella sorta di operazioni "non standard" che erano divenute molto comuni. Ad esempio, per accelerare la visualizzazione dei dati sullo schermo, la maggior parte dei programmi scriveva i dati direttamente in uno schermo mappato in memoria. In un sistema operativo operante in modalità protetta con più programmi in esecuzione, questa operazione provoca il blocco del sistema.

Ma un sistema operativo multiutente operante in modalità protetta doveva anche assicurare un certo livello di sicurezza. Nel mercato dei mainframe e dei minicomputer, il computer può essere utilizzato contemporaneamente da due o più utenti. Per motivi di sicurezza è importante che un utente non possa accidentalmente scrivere o leggere i dati o il codice del programma di un altro utente. Il 286 è stato progettato in modo da partire in modalità reale (emulando l'8086) e dopo il passaggio in modalità protetta non poteva più tornare indietro. Se ciò fosse stato consentito, un programmatore poteva scrivere un programma in grado di accedere a un altro programma. Per questo motivo è molto difficile scrivere sistemi operativi operanti in modalità protetta in grado di supportare l'esecuzione dei nuovi programmi dedicati alla modalità protetta e comunque in grado di eseguire i più vecchi programmi DOS.

Le critiche

Per la natura un po' oscura del PC AT IBM, il 286 potè essere commutato dalla modalità protetta alla modalità reale tramite alcuni circuiti esterni. L'operazione veniva eseguita disattivando la CPU e facendo in modo che il controller della tastiera inviasse un segnale di reset hardware che, in pratica, riaccendeva la CPU. Ora che la CPU era ancora in funzione, il codice rimasto in memoria rilevava che la CPU si trovava ancora in modalità reale ed era pertanto in grado di eseguire un programma DOS. L'intero processo era estremamente lento, richiedendo molto più tempo di una commutazione di modalità dell'80386.

Questo trucco fu impiegato dalle prime versioni di OS/2 per eseguire programmi DOS. Poiché l'80286 non era in grado di commutare direttamente la modalità di funzionamento, questo microprocessore fu oggetto di molte critiche. Gli ingegneri Intel avevano probabilmente terminato la maggior parte della fase progettuale dell'80286 contemporaneamente all'uscita del PC IBM. I ricercatori di mercato IBM prevedevano di vendere dai 50 mila ai 100 mila PC nei primi 2 anni ... invece ne vennero venduti milioni.

Il 286 aggiungeva anche un certo numero di nuove istruzioni. La maggior parte di esse aveva a fare con l'impostazione e il controllo delle operazioni in modalità protetta. Inoltre molte istruzioni furono ottimizzate in modo da richiedere un minor numero di cicli macchina. Infine, le istruzioni che leggevano o scrivevano sulla memoria vennero accelerate anche di due o di tre volte.

3.4 L'80386 a 32 bit

Nel 1985 Intel annuncia l'80386. Alla fine del 1986, Compaq annuncia il primo sistema compatibile IBM basato sul microprocessore 80386. La prima macchina 386 IBM fu annunciata nella primavera del 1987 con le nuove macchine della serie PS/2 e il sistema operativo OS/2 operante in modalità protetta. Nel 386 vi erano molte nuove funzionalità:

- indirizzi e dati a 32 bit;
- modalità virtuale 8086;
- memoria virtuale;
- supporto di debug integrato;
- nuove istruzioni;
- cicli più veloci;
- maggiore velocità di clock.

Negli anni successivi il discorso della compatibilità divenne sempre più importante. Le vendite dei sistemi Microchannel IBM (PS/2) non andavano così come ci si sarebbe aspettato. Anche se tutto il software rimaneva compatibile, non vi era una grande disponibilità di schede hardware e le poche che esistevano erano molto costose. Si trattò anche dei primi PC IBM con dischi da 3.5". Tutte le macchine precedenti utilizzavano vari formati di dischi da 5.25". Anche se vi sono stati problemi, ogni nuova unità dischi era in grado di leggere i dati in tutti i formati precedenti. Questo ovviamente non era possibile nel caso dei floppy da 3.5". Il progetto del PS/2 non consentiva l'inserimento di un'unità floppy da 5.25" all'interno del sistema. IBM offriva un'unità da 5.25" esterna che però non era disponibile nel momento in cui venne annunciato il sistema PS/2. Al giorno d'oggi quasi tutti sono passati al formato da 3.5" che si dimostra più affidabile e trasportabile, ma il passaggio ha richiesto molti anni. In ogni caso, il mercato ha respinto in generale il passaggio al nuovo tipo di bus Microchannel. Nel 1988, un consorzio dei produttori di hardware ha creato il bus EISA. Ma tale bus ebbe successo solo nel caso delle macchine più potenti: in altre parole, i clienti che richiedevano le macchine più potenti acquistavano qualsiasi cosa che aumentasse la velocità del sistema. Gli altri si accontentavano della compatibilità a basso costo.

Alla fine del 1987, IBM e Microsoft produssero la versione 1.0 di OS/2. Negli anni successivi vennero prodotte le versioni 1.1, 1.2 e 1.3. Nessuna di queste versioni ebbe molto successo. I motivi possono essere molti. OS/2 1.x era in grado di eseguire solo alcuni programmi DOS e solo un programma DOS per volta. Inoltre i programmi DOS risultavano molto più lenti rispetto all'originale. Anche se era possibile eseguire in multitasking più programmi OS/2, era la necessaria una riscrittura di tali programmi specifica per OS/2. Inoltre non vi erano molti driver per l'ampia gamma di prodotti disponibili sul mercato. Nel frattempo, sul mercato apparve una serie di prodotti che aumentavano la memoria disponibile in DOS e/o consentivano di eseguire contemporaneamente più programmi DOS. Alcuni di questi, come DesqView di Quarterdeck erano già disponibili da tempo ma attendevano l'uscita di una piattaforma più affidabile come era il 386.

Inoltre il 336 consentì lo sviluppo di una nuova classe di prodotti: i debugger operanti in modalità protetta. Il più noto di questi fu Soft-ICE di Nu-mega Technologies. Il debugger Soft-ICE può impostare breakpoint utilizzando i registri di debug del 386. Precedentemente, questo tipo di breakpoint era disponibile solo utilizzando costosi circuiti ICE (In-Circuit Emulator).

Nel 1988, Intel annuncia l'80386SX. Il 386SX è come un comune 386, ribattezzato 386DX, tranne per il fatto che il bus dati esterno è a 16 bit invece che a 32 bit. Inoltre il 386SX aveva solo 24 linee per gli indirizzi invece di 32 e questo semplificava la riprogettazione delle schede madri per l'80286 in modo che potessero accettare l'80386SX. Le macchine con 386SX erano più lente rispetto alle macchine con 386DX ma erano più economiche, essendo molto simili ai 286. Ma il 386SX era in grado di eseguire tutto il software per 386, inclusi i gestori di memoria e i debugger.

3.5 L'80486: l'architettura RISC

Nel 1989 Intel annuncia il microprocessore 80486. Il vantaggio di questo nuovo chip era dovuto alla sua velocità e al fatto che non richiedeva alcuna riprogrammazione del codice. Il 486 introduce solo 6 nuove istruzioni, nessuna delle quali viene molto impiegata all'interno dei programmi applicativi. Il 486 fu il primo chip della famiglia dotato di memoria cache interna. L'aspetto interno del 486 è infatti molto diverso rispetto ai precedenti chip 80x86. Le funzioni centrali della CPU sono implementate quasi come in un chip RISC. Grazie a questo, il punto di ottimizzazione fra istruzioni sempre più potenti e complesse e istruzioni semplici ma veloci iniziò a spostarsi verso queste ultime. Il 486 era anche dotato di un coprocessore interno per calcoli in virgola mobile e questo rendeva il chip molto più veloce del corrispondente 386 con coprocessore matematico 387.

3.6 L'80586

Nel 1993 Intel annuncia il Pentium. Per motivi marketing e legali Intel ha deciso di non impiegare il nome 80586. Il Pentium è molto simile al 486 ma è dotato di due pipeline per le istruzioni, una cache di maggiori dimensioni e di altre funzionalità hardware che hanno lo scopo di aumentarne le prestazioni. A parte questo vi erano alcune nuove istruzioni, principalmente destinate all'impiego da parte dei sistemi operativi. Ma la presenza delle due pipeline nel Pentium ha notevolmente aumentato la possibilità di ottimizzare i programmi precedenti o di scrivere nuovi programmi con prestazioni molto più elevate.

La competizione

Nel 1994, uno sforzo congiunto di IBM, Motorola e Apple ha portato allo sviluppo della CPU PowerPC. Questa nuova famiglia di chip è in diretta concorrenza dei chip Intel. Per una descrizione di questo chip e un confronto con il Pentium si consulti il Capitolo 20.

3.7 Il P6

L'ultimo nato della famiglia 80x86 è chiamato P6. La sua architettura interna non è più un segreto. Forse alla Intel saranno più misteriosi con il prossimo chip e lo chiameranno P007!

Tabella 3.1 Aumento delle prestazioni per le CPU 80x86.

	Anno	MIPS	MHz iniziali	Transistor
8088	1979	0,33	4,77	29.000
80186	1981	0,7	5,0	100.000
80286	1982	1,5	6,0	134.000
80386	1985	5,0	16	275.000
80486	1989	20	25	1,2 milioni
Pentium	1993	100	60/66	3,1 milioni
P6	1995	200	133	6,0 milioni
P7*	1997/98	500	250	12 milioni

Note:
* = stima
MIPS = Milioni di Istruzioni per Secondo
MHz = clock in milioni di cicli per secondo
Transistor = numero di transistor inclusi nel progetto della CPU

Parte seconda

LA FAMIGLIA 80X86

Capitolo 4

Architettura e istruzioni dell'8086

In questo capitolo viene affrontato l'argomento delle istruzioni e dell'architettura interna dei microprocessori 8088/8086. L'architettura di una CPU è costituita dai suoi registri interni, dal modo in cui tali registri vengono impiegati, dalla quantità di memoria alla quale è possibile accedere e dal modo in cui vengono codificate le istruzioni.

Questo capitolo può sembrare un po' troppo esteso specialmente per chi non è già preparato sull'argomento. Chi abbia già usato l'assembler 80x86 può procedere rapidamente nella lettura di questo capitolo mentre gli altri è bene che leggano il capitolo con la maggiore attenzione possibile.

4.1 L'architettura dell'8088

Per un programmatore, il microprocessore 8086 e 8088 sono identici, tranne per alcune lievi differenze di sincronizzazione. Il trasferimento di dati dalla CPU alla memoria e viceversa è identico sia sull'8088 che sull'8086. Semplicemente tale operazione richiede quattro cicli in più sull'8088 poiché il trasferimento viene eseguito in due tempi, ovvero un byte per volta.

Il chip più utilizzato è stato certamente l'8088, il "motore" del PC IBM originale e questo è il motivo per cui il capitolo si concentra proprio su questo chip. In particolare verranno trattate le istruzioni più utilizzate e le possibilità di ottimizzazione disponibili. Una guida completa al linguaggio assembler contenente tutte le istruzioni della CPU e tutte le direttive assembler occuperebbe troppo spazio e non rientrerebbe negli scopi di questo testo.

I registri

L'8088 contiene i dati in registri di 16 bit. Si può pensare ai registri come a scatole che contengono un numero fisso di cifre. L'8088 ha 14 registri: AX, BX, CX, DX, DI, SI, BP, SP, IP, CS, DS, ES, SS e un registro per flag (vedere la Figura 4.1).

Ma che cos'hanno di così importante questi registri? I registri sono l'unico luogo in cui possono svolgersi le operazioni della CPU. Tutte le somme, le sottrazioni, i confronti e così via devono aver luogo in un registro. Ad esempio, per eseguire una somma in una locazione di memoria, la CPU deve prima leggere il contenuto della locazione di memoria, eseguire la somma e quindi scrivere i dati di nuovo in memoria. Per molti versi i registri non sono altro che variabili (variabili a 16 bit nell'8088 e nell'8086), ed è chiaro che le operazioni svolte sui valori già contenuti nei registri sono molto più veloci rispetto a quelle svolte su variabili contenute in memoria che devono essere trasferite nei registri e poi di nuovo in memoria.

Un'ultima annotazione: alcune istruzioni utilizzano un registro aggiuntivo, senza nome, che si trova all'interno della CPU e non richiedono il caricamento di dati in uno dei registri precedentemente nominati.

Segmentazione

Lo schema di indirizzamento della memoria utilizzato dall'8088 si basa sull'uso di due registri che forniscono un indirizzo di memoria. Molti computer utilizzano per un indirizzo un singolo registro. Entrambi i metodi presentano vantaggi e svantaggi che però non verranno discussi in questa sede. Semplicemente si parlerà del funzionamento di questa tecnica. Lo spazio totale di indirizzamento, in byte, di una CPU si basa sul numero di linee di indirizzi che connettono fisicamente la CPU ai chip di memoria. Nel caso dell'8088 vi sono 20 linee di indirizzi. Elevando 2 all'esponente 20 si ottiene un totale di 1048576 byte ovvero 1 MB. Poiché i registri dell'8088 contengono solo 16 byte (ovvero possono contenere valori compresi tra 0 e 65535, 64 KB) i progettisti dell'8088 dovevano fornire un metodo per indirizzare tutto lo spazio di indirizzamento, ovvero 1 MB. La soluzione consiste nell'impiegare una combinazione di due registri, uno dei quali fornisce i 16 bit più alti dell'indirizzo composto da 20 bit mentre l'altro fornisce i 16 bit più bassi di tale indirizzo (i 12 bit centrali si sovrappongono).

Per puntare a un determinato indirizzo, l'8088 riunisce il valore in un registro di segmento (vedere di seguito) con un valore di offset. L'origine da cui proviene il valore di offset verrà discussa dettagliatamente più avanti ma può essere una combinazione di un valore di spostamento (una costante), un registro base (BX o BP) o un registro indice (SI o DI).

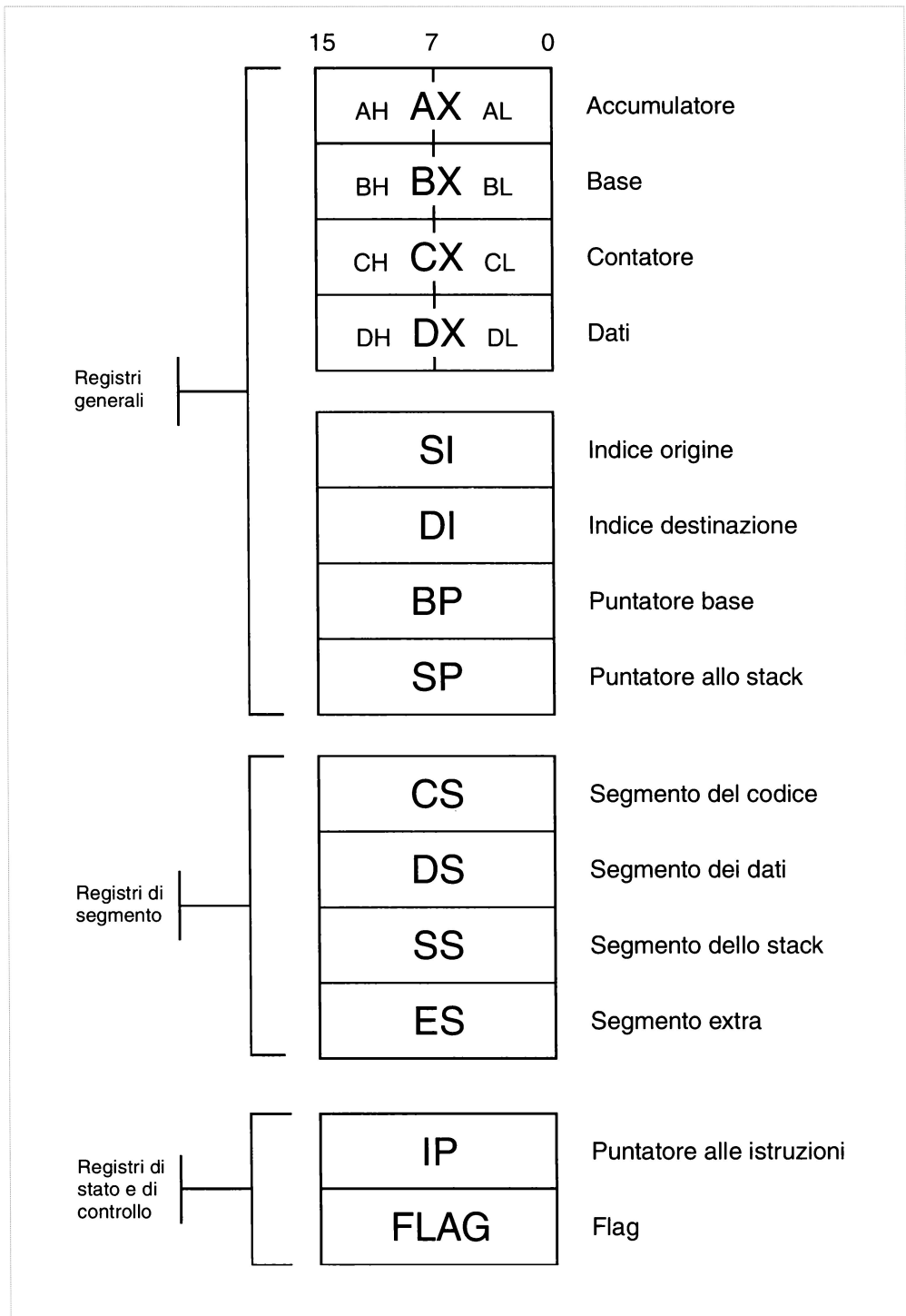



Figura 4.1 I registri dell'8088.

Nell'8088 un paragrafo di memoria è definito come un blocco contiguo di 16 byte all'interno di una strutturazione della memoria in blocchi da 16 byte. Dunque i registri di segmento forniscono il numero del paragrafo iniziale di un segmento di memoria. L'indirizzo di offset è la posizione alla quale ci si deve spostare partendo dall'inizio del segmento (vedere Figura 4.2).

Ogni segmento può essere lungo solo 64 KB (a causa della limitazione a 16 bit dei registri utilizzati per gli offset). Vi sono varie tecniche disponibili per scrivere programmi con più di 64 KB di codice o di dati. Alcune di queste verranno discusse più avanti. Inoltre, sull'80386 e i chip successivi esistono modalità che consentono di impiegare offset composti da 32 bit in modo da fornire segmenti di 4 GB.

La Tabella 4.1 mostra i segmenti utilizzati per le varie operazioni svolte dall'8088.

Indirizzi effettivi

Gli offset e gli indirizzi effettivi sono più o meno la stessa cosa. Il termine indirizzo effettivo viene normalmente utilizzato quando si fa riferimento al processo di calcolo di un indirizzo da utilizzare all'interno di un'istruzione: la somma di un valore di scostamento (una costante), di un registro base e di un registro indice. L'offset è il numero a 16 bit risultante che specifica la locazione di memoria rispetto all'inizio del segmento. Per alcuni esempi si consulti la parte relativa all'istruzione MOV.

Superamento dei limiti dei segmenti

Quando si deve eseguire un'operazione che utilizza un segmento diverso da quello standard (come illustrato nella Tabella 4.1), si devono superare i limiti del segmento. Come si può vedere dalla tabella, solo alcune operazioni consentono di accedere ad altri segmenti. Per alcuni esempi si può consultare la parte relativa all'istruzione MOV.

Tabella 4.1 Selezione dei registri di segmento.

Operazione	Segmento standard	Altri segmenti	Offset
Fetch dell'istruzione	CS	nessuno	IP
Stack	SS	nessuno	SP
Origine della stringa	DS	CS, ES, SS	SI
Destinazione della stringa	ES	nessuno	DI
BP come registro base	SS	CS, DS, ES	qualsiasi EA
BX come registro base	DS	CS, DS, ES	qualsiasi EA
SI o DI come indici	DS	CS, DS, ES	qualsiasi EA
Altre variabili in memoria	DS	CS, DS, ES	qualsiasi EA

Note: EA = indirizzo effettivo

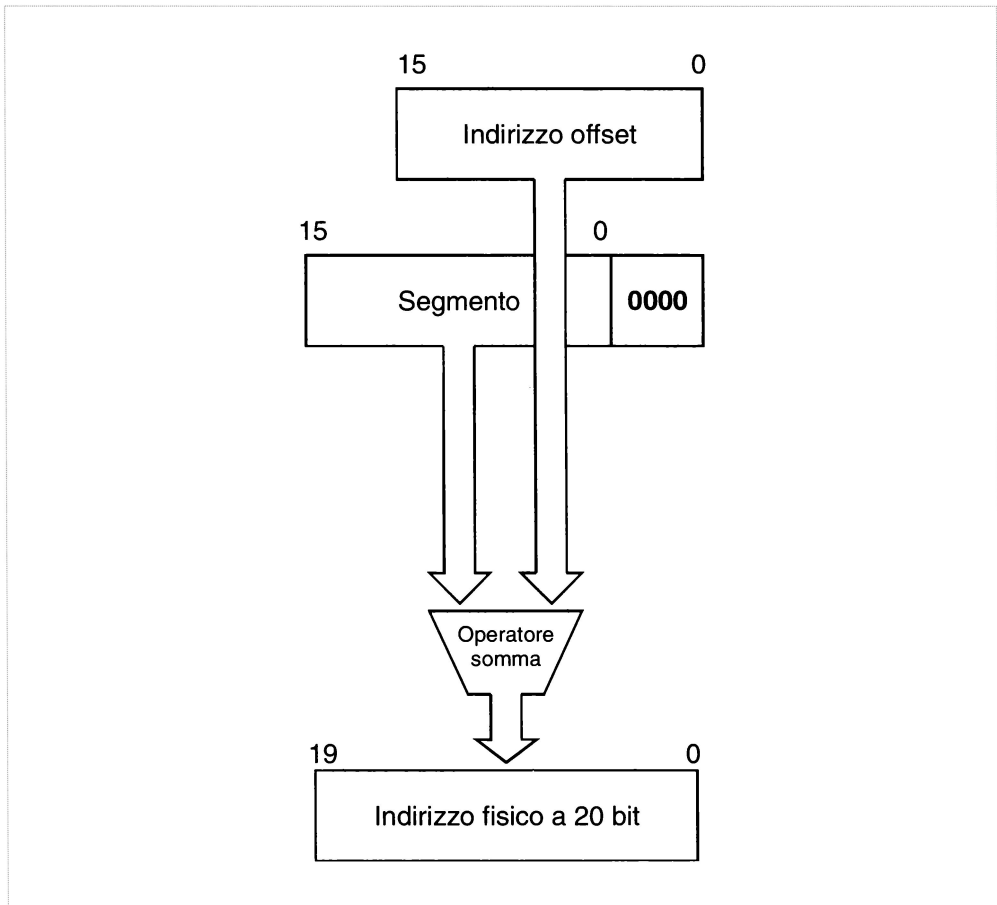



Figura 4.2 Generazione degli indirizzi fisici.

Gli stack

Uno stack è una struttura di dati formata da un blocco di memoria utilizzato in modo First-In-Last-Out, ovvero come una pila di piatti a un ristorante. La CPU conserva un puntatore all'ultimo elemento utilizzato nello stack. Nell'8088 lo spazio dello stack viene allocato una word (16 bit) per volta. L'operazione di inserimento di una word nello stack viene chiamata PUSH mentre l'estrazione di una word dallo stack è chiamata POP. L'inserimento di una word nello stack provoca un decremento del puntatore allo stack di 2 unità e la scrittura dell'elemento all'indirizzo puntato dal puntatore allo stack. Quando una word viene estratta dallo stack, prima viene copiata la word che si trova all'indirizzo puntato dal puntatore allo stack e quindi viene incrementato di due unità il puntatore. Il puntatore allo stack è sempre SS:SP, dove SS è il segmento

dello stack e SP è il puntatore dello stack. Nessuna operazione sullo stack modifica il registro SS (ovvero il registro che conserva il segmento dello stack).

Un sistema può conservare anche più stack ma solo uno stack può essere attivo in un determinato momento. SS:SP punta sempre alla cima dello stack. Per cambiare stack basta inserire nuovi valori nei registri SS e SP. L'operazione viene eseguita dal sistema operativo ovvero in genere l'utente non se ne deve preoccupare. Per esempi di operazioni sullo stack si vedano le istruzioni CALL e RET nella Figura 4.3 e le istruzioni PUSH e POP nella Figura 4.7.

I registri generali AX, BX, CX e DX

Questi quattro registri sono registri a 16 bit di utilizzo generale. La maggior parte delle istruzioni di base (somma, sottrazione, confronto e così via) può operare su uno qualsiasi di questi registri. Tali registri possono essere manipolati anche solo 8 bit alla volta, divenendo così 8 registri a 8 bit. In questo caso, quando si fa riferimento al byte inferiore, la "X" viene sostituita dalla lettera "L" mentre quando si fa riferimento al byte superiore, la "X" viene sostituita dalla lettera "H". Ad esempio, il registro AX è composto da AL e AH. Ognuno di questi registri ha alcuni utilizzi particolari.

AX è l'accumulatore. Molte istruzioni hanno un formato abbreviato che utilizza i registri AL o AX. Altre istruzioni operano su AL o AX considerandolo un registro implicito (ad esempio le moltiplicazioni, le divisioni, le istruzioni su stringhe e decimali BCD e infine le istruzioni matematiche). I registri impliciti sono registri che

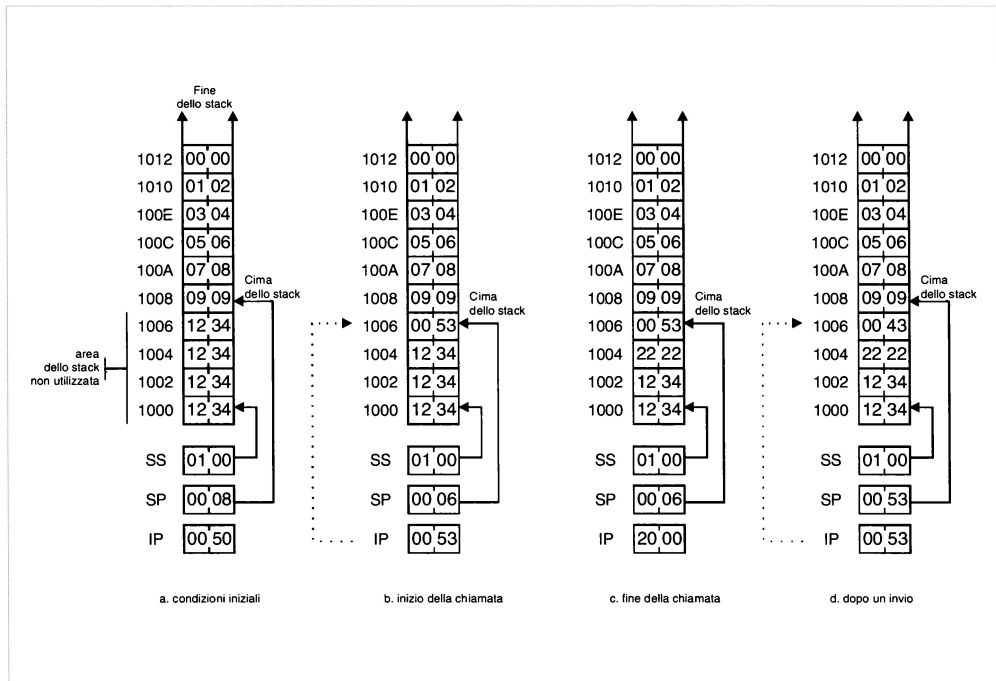



Figura 4.3 Esempi di operazioni sullo stack.

vengono utilizzati automaticamente senza essere nominati. La descrizione delle istruzioni riporta comunque ogni eventuale registro implicito utilizzato. Il nome "accumulatore" era originariamente utilizzato in molti computer al posto della parola "registro". Questo è probabilmente dovuto all'evoluzione delle macchine di somma meccanica in dispositivi elettronici.

BX è un registro base che può essere utilizzato come offset in un indirizzo di memoria. Ad esempio, per trasferire il contenuto del byte all'indirizzo 6 nel registro AL si usano le istruzioni:

```
mov   bx, 6
mov   al, [bx]
```

Le parentesi quadre che circondano BX indicano che il valore contenuto in BX (in questo caso 6) deve essere considerato come un indirizzo di memoria. Al termine di queste due istruzioni, AL conterrà una copia di ciò che si trovava all'indirizzo di memoria 6 all'interno del segmento dati corrente.

Il registro CX è utilizzato come contatore per i cicli e le operazioni sulle stringhe. Ad esempio, per sommare 5 byte a partire dall'indirizzo 6 e inserire il risultato nel registro AL si utilizzano le seguenti istruzioni:

```
 mov   bx, 6    ; indirizzo iniziale
 mov   al, 0    ; inizializza count a 0
 mov   cx, 5    ; carica in cx il numero di cicli da eseguire
adder:
 add   al, [bx] ; somma un byte
 add   bx, 1    ; punta al byte successivo
 loop  adder    ; sottrae 1 da CX e continua il ciclo finché CX=0
```

Il registro DX è utilizzato come word alta nelle moltiplicazione e divisioni a 32 bit. Quando si moltiplicano due valori a 16 bit, il risultato può essere un valore a 32 bit. Il risultato può quindi richiedere due registri a 16 bit. A tale scopo vengono sempre utilizzati i due registri DX e AX . Ad esempio, per moltiplicare 1000 per 2000 si utilizzano le seguenti istruzioni:

```
mov   bx, 1000
mov   ax, 2000
mul   bx              ; (AX operando implicito)
```

Il risultato, 2.000.000 (1E8480h) sarà dunque contenuto nel registro DX (la word alta ovvero 001Eh) e AX (8480h).

I registri base indice BP, DI, SI

Si tratta di tre registri base o indice a 16 bit ma anche di registri di utilizzo generale. Anche se è possibile accedervi 16 bit per volta, possono essere impiegati più o meno come gli altri registri di utilizzo generale. Il registro BP, è un registro base che viene spesso utilizzato come offset per lo stack. Normalmente i compilatori per linguaggi di alto livello impostano BP in modo da puntare allo stack per ottenere i parametri passati da una procedura chiamante. Inoltre nello stack è possibile allocare spazio per

variabili locali. I registri indice DI e SI sono utilizzati come puntatori alle aree di memoria che fungeranno da origine e destinazione delle operazioni sulle stringhe.

L'uso delle parole "base" e "indice" ha significato solo nel senso che quando si accede alla memoria si riunisce il contenuto di un registro base e di un registro indice. A parte questo, il loro utilizzo è praticamente identico. L'altra differenza è dovuta al fatto che il registro BP utilizza normalmente il segmento che contiene lo stack.

Registri speciali

Vi sono vari gruppi di registri che hanno un utilizzo speciale. Alcune operazioni della CPU utilizzano automaticamente questi registri e dunque non è richiesto alcun intervento da parte del programmatore.

Registri di segmento: DS e ES

I registri di segmento sono utilizzati per conservare l'indirizzo iniziale di un segmento di memoria. Il registro DS (Data Segment) contiene l'indirizzo iniziale del segmento dati. Il registro ES (Extra Segment) contiene l'indirizzo iniziale del segmento extra. Tale segmento può essere utilizzato per ogni tipo di dati e in particolare deve essere utilizzato per alcune operazioni sulle stringhe che verranno descritte più avanti.

Si ricordi che il valore conservato nel registro di segmento non è esattamente l'indirizzo iniziale ma corrisponde solo ai 16 bit superiori dell'indirizzo che è composto da 20 bit. Per convertire il valore del registro di segmento in un indirizzo fisico, tale valore deve essere moltiplicato per 16 (o spostato a sinistra di 4 bit).

Il registro del segmento di codice e il puntatore all'istruzione: CS e IP

Il registro del segmento di codice (CS) punta al segmento che contiene il codice in esecuzione. Il puntatore all'istruzione (registro IP) contiene l'offset, nel segmento di codice, che punta all'istruzione attualmente in esecuzione. La combinazione dei registri CS:IP (Figura 4.4) viene utilizzata per puntare alla prossima istruzione che deve essere eseguita.

Il registro del segmento dello stack e il puntatore allo stack: SS e SP

Il registro del segmento dello stack (SS) contiene l'indirizzo iniziale del segmento che contiene lo stack. Il puntatore allo stack (registro SP) contiene l'offset, nel segmento dello stack, che punta alla cima dello stack. Se si uniscono questi due registri, SS:SP, si ottiene un puntatore alla cima dello stack. Lo stack viene impiegato per conservare gli indirizzi restituiti dalle procedure, per salvare i registri e per le variabili locali. A tale proposito si consulti la sezione precedente che descrive il funzionamento dello stack.

Il registro dei flag

Il registro dei flag (noto anche con i nomi di "word di stato" e "codici di condizione") è formato da 16 singoli bit, ognuno dei quali ha un proprio significato (vedere Figura 4.5). Alcuni di essi possono essere impostati, cancellati o verificati singolarmente o in gruppi. Nei microprocessori 8088 e 8086 vengono utilizzati solo 10 flag. Altre CPU 80x86 utilizzano più bit. La Tabella 4.2 descrive, in termini generali, il significato di ognuno di questi flag. La comprensione e l'uso dei flag è una delle differenze principa-

li che sussistono fra il linguaggio assembler e i linguaggi di alto livello. I compilatori gestiscono automaticamente tutti i dettagli dell'impostazione e verifica dei flag. Ad esempio, quando si devono sommare due numeri, se il risultato è troppo esteso per la destinazione viene impostato il flag di overflow. A questo punto è possibile utilizzare l'istruzione di salto condizionale "JO" (jump on overflow) che salta a una routine di gestione dell'errore. Non tutte le istruzioni alterano i flag. Dopo aver studiato il set di istruzioni, si può credere che le istruzioni che modificano i flag e quelle che non li modificano siamo stati scelte un po' a caso. Ad esempio, quando si sommano due numeri e il risultato è 0, viene impostato il flag Zero (ZF).

Tabella 4.2 I flag del microprocessore 8088/8086.

Bit 0	**CF - Carry Flag** Questo flag viene impostato quando vi è un riporto in ingresso o in uscita nel bit più alto del risultato. Per cancellare il riporto si possono utilizzare scorrimenti e rotazioni. Può essere impostato (STC), cancellato (CLC) o complementato (CMC) direttamente.
Bit 1	Riservato, sempre uguale a 1.
Bit 2	**PF - Parity Flag** Questo flag viene impostato quando un risultato ha parità pari. La parità viene verificata solo nel byte inferiore.
Bit 3	Riservato
Bit 4	**AF - Auxiliary Flag** Questo flag viene impostato quando vi è un riporto nel nibble inferiore verso il nibble superiore o viceversa. Questo flag viene impostato o cancellato da varie istruzioni aritmetiche ed è utilizzato dalle istruzioni aritmetiche BCD.
Bit 5	Riservato.
Bit 6	**ZF - Zero Flag** Questo flag viene impostato quando il risultato di un'operazione è uguale a 0.
Bit 7	**SF - Sign Flag** Questo flag viene impostato quando il risultato di un'operazione ha attivato il bit superiore. In questo caso i numeri con segno sono negativi.
Bit 8	**TF - Trap Flag** Questo flag è chiamato anche Single Step Flag. L'impostazione di questo bit pone il processore in modalità Single Step. I debugger utilizzano la modalità Single Step per il debug del programma.
Bit 9	**IF - Interrupt Flag** L'impostazione di questo flag consente al processore di rispondere agli interrupt provenienti dall'esterno. La cancellazione di questo flag disattiva tali interrupt. Gli interrupt non mascherabili (NMI) non vengono influenzati da questo flag.
Bit 10	**DF - Direction Flag** L'impostazione di questo flag provoca un autodecremento delle istruzioni sulle stringhe. La cancellazione del flag provoca un autoincremento delle istruzioni sulle stringhe.
Bit 11	**OF - Overflow Flag** Questo flag viene impostato quando un'istruzione aritmetica ha perso un bit significativo a causa di un overflow.
Bit 12	Riservato.
Bit 13	Riservato.
Bit 14	Riservato.
Bit 15	Riservato.

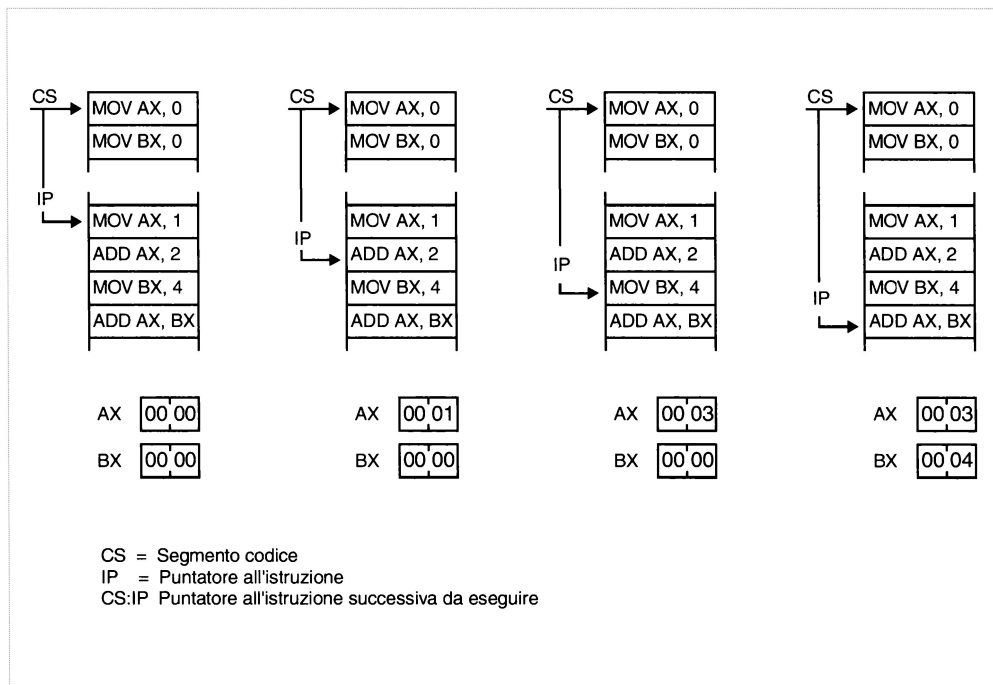



Figura 4.4 Il funzionamento dei registri CS:IP.

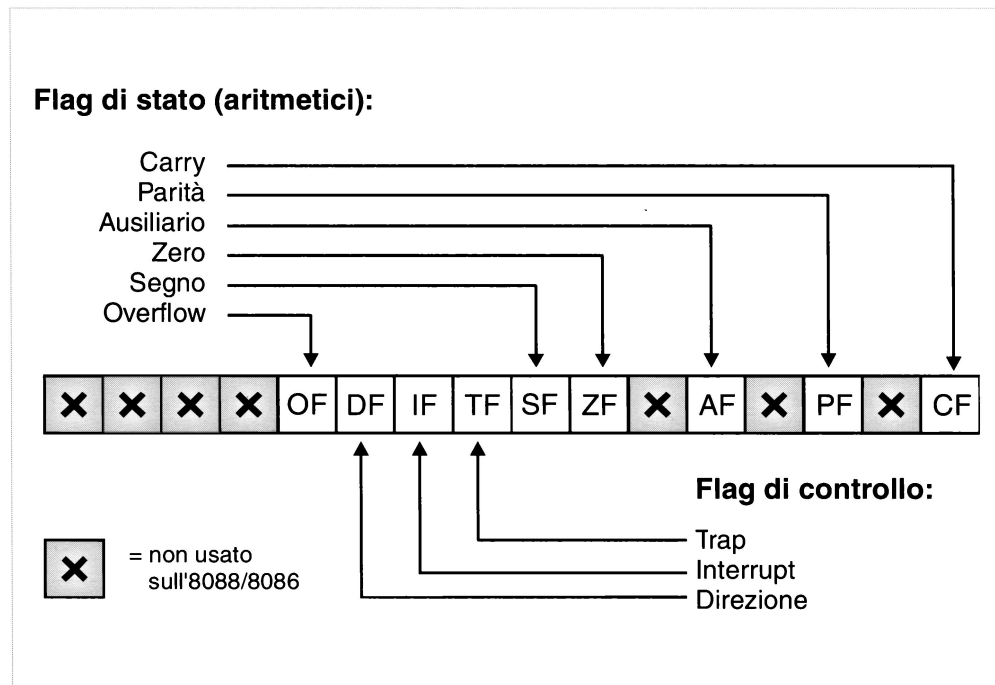
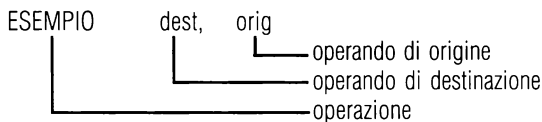



Figura 4.5 Diagramma dei flag con le posizioni dei bit.

Se però si copia uno 0 in un registro, il flag Zero (ZF) non viene modificato. Alcune istruzioni modificano solo alcuni dei flag e non altri. Tutto ciò ha però un senso che verrà descritto più avanti in questo stesso capitolo.

4.2 Il set di istruzioni dell'8088

Questa sezione introduce l'uso delle più comuni istruzioni dell'8088. Si tratta semplicemente di una breve descrizione che indica solo ciò che le istruzioni fanno senza entrare nei dettagli del modo o del perché tali istrùzioni debbano essere utilizzate per ottenere il risultato desiderato. L'ultima sezione di questo capitolo contiene alcuni esempi del modo in cui possono essere utilizzate varie combinazioni di queste istruzioni per scopi specifici. Dai semplici esempi mostrati in precedenza si sarà notato che le istruzioni assembler utilizzano una notazione comune.

I processori Intel utilizzano la seguente notazione:

Non tutte le istruzioni hanno sia l'origine che la destinazione. Alcune istruzioni hanno un solo operando che funge sia da origine che da destinazione. Altre istruzioni non hanno operandi espliciti ma solo operandi impliciti. Questa "irregolarità" può anche preoccupare ma è fonte di una gran parte delle opportunità di ottimizzazione dei programmi. Occorre sempre ricordare che se vi sono due operandi, quello a sinistra è la destinazione. Nella prossima sezione verranno introdotti i vari tipi di istruzioni.

MOV

L'istruzione MOV copia il valore dall'operando di origine nell'operando di destinazione. In pratica l'istruzione si occupa di copiare dati. Ad esempio,

```
mov   ax, 1
```

Questa istruzione inserisce il numero 1 nel registro AX. In pratica equivale all'istruzione:

```
mov   ax, 1   ; ax=1
```

Vi sono varie combinazioni di origini e destinazioni. L'origine può essere un valore immediato (una costante), un registro o un indirizzo di memoria. La destinazione può essere un registro o un indirizzo di memoria. Questo significa che vi sono cinque combinazioni di origini e destinazioni, descritte nel seguente listato:

```
mov   reg, reg
mov   reg, cos
mov   reg, mem
mov   mem, reg
mov   mem, cos
```

dove

reg = registro: un registro generale a 8 o 16 bit
cos = costante: un valore costante a 8 o 16 bit
mem = memoria: l'indirizzo di un byte o di una word in memoria

Gli indirizzi di memoria possono essere un offset all'interno di un segmento oppure si possono basare su un indirizzo contenuto in un registro. In questo secondo caso è possibile sommarvi un valore di scostamento costante. I registri utilizzabili per conservare un indirizzo sono BX, BP, SI e DI.

Da registro a registro:

```
MOV   ax, bx
MOV   al, bh
```

Da costante a registro:

```
MOV   ax, 2
```

Da costante a memoria:

```
MOV   dati_mem, 0
MOV   byte ptr [bx], 1
MOV   word ptr [si+1], 2
```

Da memoria a registro:

```
MOV   ax, [bx]
MOV   al, [di-1]
MOV   cx, dati_mem
```

Da registro a memoria:

```
MOV   [bx], ax
MOV   [di-1], a1
MOV   dati_mem, cx
```

Si può aver notato che nella categoria "Da costante a memoria" vi sono due nuovi termini: "byte ptr" e "word ptr". Non si tratta di istruzioni per la CPU ma per l'assembler. In pratica qualificano il tipo dei dati. L'istruzione "MOV [BX], 1" è ambigua. Nell'indirizzo contenuto nel registro BX occorre spostare un byte o una word contenente il valore 1? Non vi è alcun modo per saperlo e dunque l'assembler, se non si specifica il tipo dei dati, genererà un errore.

Ci si potrebbe chiedere: "Ma perché non è richiesto il tipo dei dati nell'istruzione MOV dati_mem, 0?". La risposta è che la variabile dati_mem deve essere stata definita o dichiarata in qualche altro punto del programma. Dunque l'assembler conosce già il tipo dei dati. Questo argomento verrà descritto in dettaglio nel Capitolo 6.

MOV nei registri di segmento

Lo spostamento di un valore in un registro di segmento è leggermente diverso rispetto allo spostamento di dati in altri registri poiché non vi è alcun modo per inserire un

valore costante in un registro di segmento. Gli unici impieghi dell'istruzione MOV con un registro di segmento (sreg) sono:

```
mov    sreg, reg
mov    sreg, mem
mov    mem, sreg
mov    reg, sreg
```

dove:

sreg = registro di segmento: DS, ES o SP
reg = registro: un registro a 16 bit di utilizzo generale
mem = memoria: l'indirizzo di una word in memoria

Vi sono varie spiegazioni per questa limitazione ma nessuna di esse è molto convincente. L'opinione dell'autore è che sia rimasto un problema insoluto. Ecco alcuni esempi:

```
mov    ax, 40h
mov    ds, ax
mov    es, [bx]
```

Modalità di indirizzamento

I vari metodi utilizzabili per leggere e scrivere in memoria vengono chiamati modalità di indirizzamento. L'8088 consente di specificare un indirizzo utilizzando una combinazione di tre oggetti. Per la precisione, l'indirizzo utilizzato in un segmento base (registro di segmento moltiplicato per 16) più la somma di altre tre componenti:

un valore di scostamento (costante)
un registro base (BX o BP)
un registro indice (SI o DI)

Un indirizzo può essere costituito da una qualsiasi combinazione di queste tre componenti. Ecco alcuni esempi:

```
mov ax, [1234]      ; scostamento
mov ax, var1        ; scostamento (il valore di var1 è determinato dall'assembler)
mov ax, [bx]        ; base
mov ax, [si]        ; indice
mov ax, [bx+1]      ; base + scostamento
mov ax, [si+2]      ; indice + scostamento
mov ax, [bx+si]     ; base + indice
mov ax, [bx+si+3]   ; base + indice + scostamento
```

Queste sono le uniche combinazioni che l'assembler può generare per l'8088. Tuttavia, alcune espressioni che generano indirizzi di memoria possono essere diverse o più complesse. Si tratta in ogni caso di metodi diversi per generare la porzione costante (lo spostamento) dell'indirizzo. Nota: a partire dal 386 vi sono nuove modalità di indirizzamento; a tale proposito si consulti il Capitolo 7.

ADD - ADDition

L'istruzione ADD esegue una somma binaria su due operandi, inserendo poi il risultato in uno dei due operandi. Ad esempio:

```
add	ax, bx
```

somma AX e BX e memorizza il risultato in AX. In pratica si ottiene il seguente risultato:

AX = AX + BX

Dunque, anche se l'istruzione ADD utilizza due soli operandi, entrambi vengono considerati operandi sorgenti e il primo operando è anche destinazione. L'istruzione ADD può utilizzare le modalità di indirizzamento della memoria viste per l'istruzione MOV.

Molte istruzioni aritmetiche modificano i bit del registro dei flag. Ad esempio, un'istruzione ADD modifica i flag secondo quanto descritto nella Tabella 4.3.

Istruzioni aritmetiche e logiche (alu)

L'istruzione ADD appena descritta non è che un esemplare di un gruppo di istruzioni note con il nome generale di istruzioni ALU. L'Arithmetic Logic Unit (ALU) è quella porzione della CPU che esegue tutti i calcoli aritmetici e logici.

Molte delle operazioni ALU modificano i flag elencati nella tabella nella Tabella 4.3. Tali flag verranno indicati in seguito con il nome generale di flag aritmetici. Le operazioni ALU includono le istruzioni ADD, SUB, ADC, SBB, INC, DEC, NEG, CMP, AND, OR, XOR, NOT e TEST. Tali istruzioni verranno descritte di seguito.

ADC - ADd with Carry

L'istruzione ADC esegue una somma con riporto. Questo consente di utilizzare un computer con registri a 16 bit anche per eseguire operazioni aritmetiche su interi di qualsiasi dimensione.

```
adc	ax, bx	; AX = AX + BX + CF (flag carry)
```

somma di due numeri a 32 bit in quattro registri a 16 bit:

```
		; equivalente a: DX:AX = DX:AX + CX:BX
add	ax, bx	; AX = AX + DX
adc	dx, cx	; DX = DX + CX + CF
```

SUB - SUBtraction

L'istruzione SUB esegue una sottrazione binaria sui due operandi, scrivendo il risultato al posto di uno dei due operandi.

Tabella 4.3 I flag aritmetici modificati da ADD.

Il **flag carry** (CF) viene impostato quando vi è un riporto nel bit di ordine più elevato del risultato.
Il **flag di overflow** (OF) viene impostato quando si perde un bit significativo a causa di un overflow (è una situazione diversa rispetto al riporto in operazioni con segno nelle quali si perde un bit significativo prima di ogni operazione con riporto).
Il **flag zero** (ZF) viene impostato quando il risultato di un'operazione è uguale a 0.
Il **flag di segno** (SF) viene impostato quando il risultato di un'operazione imposta il bit di ordine più elevato, ovvero il numero è negativo.
Il **flag ausiliario** (AF) viene impostato quando vi è un riporto dal nibble inferiore al nibble superiore (per istruzioni BCD, descritte più avanti).
Il **flag di parità** (PF) viene impostato quando un risultato ha parità pari nel byte di ordine inferiore. La parità è il bit meno significativo della somma del numero di bit impostati, più 1.

SBB - SuBtract with Borrow

L'istruzione SBB esegue una sottrazione binaria con riporto sui due operandi, scrivendo il risultato al posto di uno dei due operandi. Se viene impostato il flag di riporto carry (CF), viene sottratta un'ulteriore unità.

```
sub     ax, bx  ; AX = AX - BX
```

Sottrazione di due numeri a 32 bit in quattro registri a 16 bit:

```
                ;equivalente a: DX:AX = DX:AX - CX:BX
sub     ax, bx  ;AX = AX - BX
sub     dx, cx  ;DX = DX - CX - CF
```

INC - INCrement

INC somma una unità al suo operando (che funge sia da origine che da destinazione). L'operando viene sempre trattato come un numero senza segno. Questa istruzione non influenza il flag carry (CF).

```
inc     ax      ; AX = AX -1
```

DEC - DECrement

DEC sottrae una unità al suo operando (che funge sia da origine che da destinazione). L'operando viene sempre trattato come un numero senza segno. Questa istruzione non influenza il flag carry (CF).

```
dec     ax      ; AX = AX + 1
```

NEG - NEGate

L'istruzione NEG sottrae l'operando dal numero 0. L'operazione è nota anche con il nome di negazione in complemento a 2. Ciò che accade è l'inversione di ogni bit e l'incremento dell'intero operando.

```
neg     ax      ; AX = 0 - AX
```

CMP - CoMPare

CMP può essere interpretata come un'istruzione di confronto ma in realtà esegue una sottrazione tranne per il fatto che non salva il risultato. Vengono infatti modificati soltanto i flag. Il risultato è comunque un confronto. In questo esempio il contenuto di AX viene confrontato con il numero 5:

```
cmp     ax, 5   ; confronta AX con 5
jl      small   ; salta se AX è minore di 5 rispetto a small
```

Operazioni sui bit: AND, OR, XOR, NOT e TEST

Le istruzioni aritmetiche sui bit eseguono operazioni bit-a-bit su operandi costituiti da registri o oggetti in memoria. La tabella seguente mostra il risultato delle operazioni di AND, OR e XOR (OR esclusivo) su due bit.

bit 1	bit 2	AND	OR	XOR
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

L'istruzione TEST esegue un'operazione AND ma modifica solo i flag senza scrivere il risultato negli operandi (è simile al comportamento di CMP in confronto a SUB). L'istruzione NOT inverte tutti i bit di un operando.

4.3 Scorrimenti e rotazioni

Esistono due tipi di scorrimenti e due tipi di rotazioni. Entrambe le operazioni possono essere eseguite sia a destra che a sinistra. In entrambi i casi, la destinazione viene fatta scorrere o ruotare del numero dei bit specificati nel secondo operando. Tale valore può essere uguale a 1 oppure essere contenuto in CL. A partire dal microprocessore 80186 è possibile eseguire scorrimenti immediati da 1 a 31 posizioni. Le istruzioni di scorrimento influenzano tutti i flag aritmetici tranne il flag ausiliario (AF). Le istruzioni di rotazione influenzano solo il flag carry (CF) e il flag di overflow (OF). Il flag

più importante in entrambi i casi è il flag carry (CF) poiché il bit che fuoriesce dall'operando finisce in questo flag; per qualche esempio si può osservare la Figura 4.6.

SHR - SHift logical Right

L'istruzione SHR fa scorrere verso destra l'operando di destinazione del numero di posizioni desiderate. Nei bit di ordine più elevato viene inserita una serie di 0. Se il valore del bit più alto cambia, viene impostato il flag di overflow, altrimenti tale flag viene cancellato. Nel flag Carry viene copiato il bit basso originale. Lo scorrimento a destra in binario equivale alla divisione per 2 (eliminando il resto).

```
shr     ax, 1    ; AX = AX / 2
```

SAR - Shift Arithmetic Right

Lo scorrimento aritmetico è leggermente diverso dal precedente per il fatto che il valore contenuto nel bit più significativo conserva il proprio valore originario. L'effetto è che il risultato ha lo stesso segno (positivo o negativo) del valore di partenza.

```
sar     ax, 1    ; AX = AX / 2
```

SHL / SAL - SHift logical Left / Shift Arithmetic Left

Le istruzioni SHL e SAL sono identiche. L'operando di destinazione viene fatto scorrere verso sinistra del numero di posizioni specificato. Nel bit di ordine più basso viene inserita una serie di 0. Se il valore del bit di ordine più elevato cambia, viene impostato il flag di overflow (OF) che, in caso contrario, viene cancellato.

```
sh1     ax, 1    ; (CF +) AX = AX * 2
```

ROR / ROL - ROtate Right / ROtate Left

Le istruzioni di rotazione eseguono esattamente l'operazione che ci si può attendere. I bit che escono da un'estremità dell'operando ruotano e quindi riappaiono all'altra estremità. Inoltre il bit ruotato viene copiato nel flag carry (CF).

RCR / RCL - Rotate Carry Right / Rotate Carry Left

Le istruzioni di rotazione con Carry sono uguali alle precedenti, tranne il fatto che il flag carry viene incluso come una parte dell'operando. Ad esempio, lavorando con registri a 16 bit, la rotazione modifica in effetti un valore a 17 bit in cui il flag carry gioca il ruolo di bit di ordine più elevato.

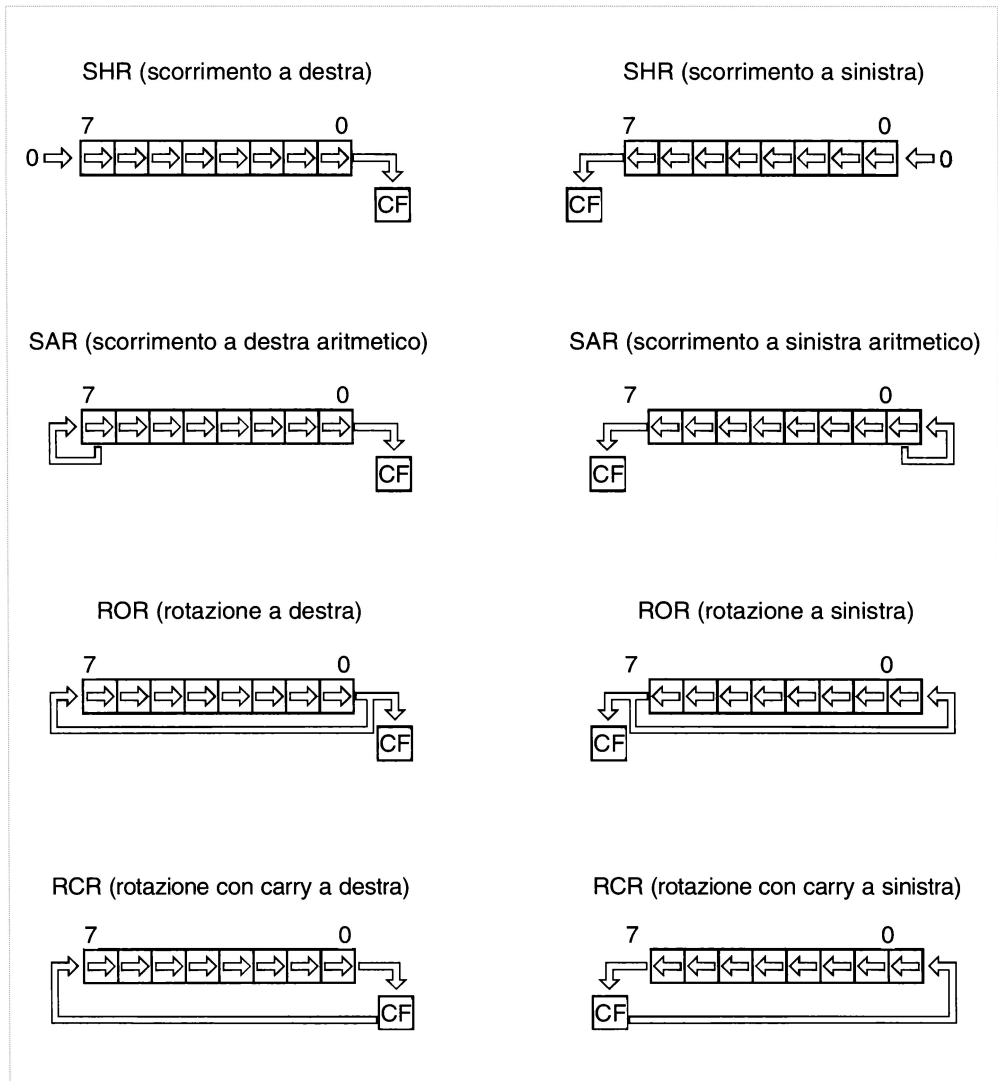



Figura 4.6 Schema degli scorrimenti e delle rotazioni a 8 bit.

PUSH

L'istruzione PUSH inserisce una word nello stack. Prima il registro SP viene incrementato di 2 unità e quindi l'operando di origine viene copiato sullo stack. L'operando può essere un registro a 16 bit o una locazione di memoria.

POP

L'istruzione POP estrae una word dallo stack. Prima la word viene copiata dallo stack all'operando di destinazione e quindi il registro SP viene incrementato di 2 unità. La

word può essere inserita in un registro a 16 bit o in una locazione di memoria (a tale proposito si consulti la Figura 4.7).

Ora si può vedere come queste istruzioni possono essere combinate fra loro.

```
push  bx     ; salva BX sullo stack
push  cx     ; salva CX
push  dx     ; salva DX
mov   ax, 10 ; AX = 10
mov   cl, 3  ; CL = 3
mov   bx, 2  ; BX = 2
push  bx     ; AX (10) sullo stack
pop   dx     ; DX = 10 (dallo stack)
shl   ax, cl ; AX   = AX * (2 * 2 * 2) --> AX = 80
             ; (tre 2 perché CL = 3)
inc   ax     ; AX   = AX    + 1 --> AX = 81
add   ax, bx ; AX   = AX    + BX --> AX = 83
sub   ax, dx ; AX   = AX    - DX --> AX = 73

pop   dx     ; ripristina DX dallo stack
pop   cx     ; ripristina CX
pop   bx     ; ripristina BX
```

Questa breve sezione di codice dimostra l'uso di varie istruzioni ma per scrivere programmi veramente utili è necessario conoscere il funzionamento di molte altre istruzioni.

4.4 Controllo del programma e salti

Le istruzioni CALL, RET e JMP modificano il flusso di esecuzione del programma trasferendo il controllo a varie procedure e subroutine.

CALL

L'istruzione CALL è molto simile a JMP tranne per il fatto che oltre a trasferire il controllo in una nuova destinazione, salva sullo stack la posizione corrente. In pratica realizza una chiamata a una subroutine. Quando la subroutine termina, esegue l'istruzione RET e il controllo torna all'istruzione che segue immediatamente l'istruzione CALL. Nel caso di chiamate near, nello stack viene inserito solo IP. Per chiamate far, nello stack vengono inseriti sia IP che CS. Ad ogni CALL deve essere associata una corrispondente istruzione RET.

```
mov   ax, 5  ; prepara AX per la chiamata
mov   bx, 2  ; prepara BX per la chiamata
call  func1  ; richiama la subroutine
```

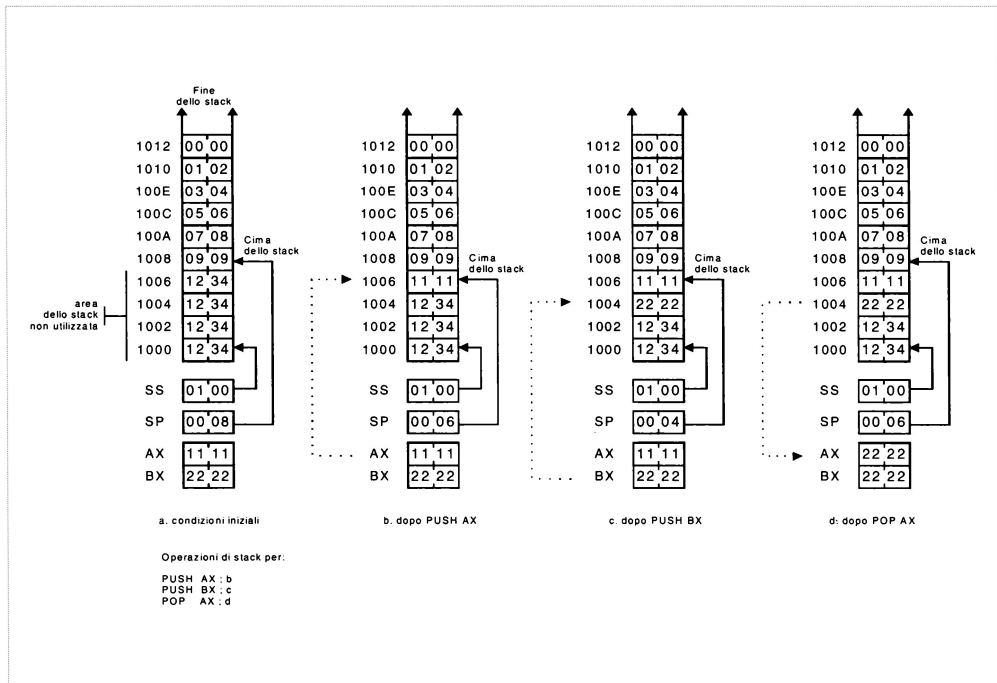



Figura 4.7 Operazioni di PUSH e POP sullo stack.

RET

L'istruzione RET riporta il controllo all'istruzione che segue immediatamente l'ultima istruzione CALL eseguita. Per fare ciò ne estrae l'indirizzo dallo stack. A una CALL near deve essere associata una RET near (RETN) e viceversa a una CALL far deve essere associata una RET far (RETF). L'operazione viene però eseguita automaticamente dall'assembler per cui basta utilizzare l'istruzione RET. Ad esempio una procedura che calcola 2 * X + Y + 1 (AX=x e BX=y) potrebbe essere:

```
func1  proc near
       sh1   ax, 1   ; moltiplica AX per 2
       add   ax, bx  ; somma BX
       inc   ax      ; somma 1
       ret           ; esce con il risultato in AX
func1  endp
```

JMP

JMP esegue un salto in un nuovo indirizzo. L'istruzione JMP può assumere diversi formati. La forma più semplice è:

```
 inc   ax
 jmp   dest
 ...
dest:
 mov   bx, ax
```

In questo modo viene eseguito un salto near (ovvero la destinazione si trova nello stesso segmento). Vi è anche un formato di salto breve. Si tratta di un formato speciale in cui la destinazione si trova a -128 o +127 byte di distanza e dunque l'istruzione richiede meno spazio (2 byte di codice invece di 3). È anche possibile eseguire salti far. Si tratta di salti ad indirizzi contenuti in altri segmenti di codice. Inoltre vi sono due tipi di salti indiretti. Il primo prevede la memorizzazione del nuovo offset di segmento in un registro:

```
jmp bx
```

L'altro tipo prevede la memorizzazione del nuovo offset (o segmento + offset) in una variabile di memoria:

```
jmp [bx]
...
jmp word ptr dest
...
new_dest:
...
dest dw new_dest
```

I vantaggi di questi tipi di salti sono legati alla possibilità di scrivere programmi in grado di riconfigurarsi. Si tratta di tecniche molto potenti, come si può vedere dal seguente esempio:

```
mov      bl, [si]       ; legge un carattere da una stringa precedentemente caricata
                        ; si assume che il carattere sia "A", "B" o "C"
mov      bh, 0          ; Inserisce uno zero nel byte alto di BX
sub      bl, 'A'        ; sottrae il valore ASCII di 'A'
jmp      jmp_tbl[bx]    ; salta al codice di elaborazione
jmp_tbl dw lbl_A        ; questa tabella contiene un array
         dw lbl_B       ; di tre puntatori near alle etichette di elaborazione
         dw lbl_C       ; di "A", "B" o "C"
lbl_A:
...                     ; codice per "A" ecc.
lbl_B:
...
lbl_C:
...
```

L'esempio precedente realizza un'istruzione CASE in assembler. Il costrutto è costituito da un'unica istruzione assembler e questo spiega il motivo della sua velocità ed efficienza. In molti casi si deve controllare l'intervallo in cui cade il parametro che

controlla la destinazione del salto (in questo esempio si tratta di BX). Per questo tipo di operazioni è necessario utilizzare salti condizionali.

Salti condizionali

I salti condizionali sono le istruzioni che consentono di prendere decisioni. Corrispondono in pratica alle istruzioni IF ... THEN dei linguaggi di alto livello. Tuttavia occorre dire che i salti condizionali sono molto più primitivi in quanto verificano solo lo stato di uno o più bit nel registro dei flag che devono essere impostati tramite altre istruzioni. Per eseguire un confronto fra interi si deve utilizzare l'istruzione CMP:

```
cmp ax, bx      ; confronta AX e BX (e attiva i flag)
jne dest        ; salta se sono diversi
mov cx, 1       ; CX = 1
dest:
```

Questa sequenza di codice equivale a:

```
IF AX = BX THEN CX = 1
```

L'istruzione JNE (Jump if Not Equal) esegue il salto se i due operandi sono differenti. L'istruzione CMP imposta infatti il flag zero (ZF). Se AX e BX sono diversi, l'istruzione CMP attiva il flag 0 (ZF). In realtà, CMP esegue una sottrazione ma non memorizza il risultato da nessuna parte, limitandosi a impostare alcuni bit nel registro dei flag.

```
  cmp   ax, bx  ; confronta AX e BX (e attiva i flag)
  jge   dest_1  ; salta se è maggiore o uguale
  mov   cx, 1   ; CX = 1
  jmp   dest_2
dest_1:
  mov  cx, 2    ; CX = 2
dest_2:
```

Questa sequenza di codice equivale a:

```
IF AX < BX THEN CX = 1 ELSE CX = 2
```

Confronti con e senza segno

Tornando agli argomenti presentati nel Capitolo 1, è importante considerare sempre la differenza fra valore numerici con e senza segno. Tuttavia questa differenza si basa solo sul modo in cui vengono utilizzati i dati. Ad esempio, il byte 0FFh può essere trattato come il numero 255 o come il numero -1. I salti condizionali sono gli strumenti più importanti che consentono di distinguere fra numeri con e senza segno. Nell'esercizio precedente, AX e BX sono stati trattati come valori con segno. Per questo

motivo è stata utilizzata l'istruzione di salto condizionale JGE. Se fosse stata utilizzata l'istruzione JAE, AX e BX sarebbero stati trattati come valori senza segno. L'uso della parola "Greater" per i numeri con segno e "Above" per i numeri senza segno è una scelta arbitraria dei progettisti della Intel. Per un elenco completo dei salti condizionali, si consulti la Tabella 4.4.

LOOP

L'istruzione LOOP consente di ripetere sezioni di codice un determinato numero di volte. Il numero di ripetizioni deve essere inserito nel registro CX prima dell'inizio del ciclo. Ad esempio, per ripetere un blocco di codice per 50 volte, si deve scrivere:

```
    mov cx, 50     ; CX = numero di cicli
loop1:
    ...
    ...
    loop loop1     ; CX = CX - 1, se CX = 0 salta a loop1
```

L'istruzione LOOP decrementa CX di una unità e quindi salta alla destinazione specificata (sempre che CX sia diverso da zero.).

Tabella 4.4 Salti condizionali.

Salto	Forma alternativa	Descrizione	
Confronti fra numeri senza segno			
ja	jnbe	Jump if Above	Jump if Not Below or Equal
jae	jnb	Jump if Above or Equal	Jump if Not Below
jb	jnae	Jump if Below	Jump if Not Above or Equal
jbe	jna	Jump if Below or Equal	Jump if Not Above
Confronti fra numeri con segno			
jg	jnle	Jump if Greater	Jump if Not Less or Equal
jge	jnl	Jump if Greater or Equal	Jump if Not Less
jl	jnge	Jump if Less	Jump if Not Greater or Equal
jle	jng	Jump if Less or Equal	Jump if Not Greater
Altri			
jc		Jump if Carry	
jnc		Jump if Not Carry	
je	jz	Jump if Equal	Jump if Zero
jne	jnz	Jump if Not Equal	Jump if Not Zero
jo		Jump if Overflow	
jno		Jump if Not Overflow	
js		Jump if Sign	
jns		Jump if Not Sign	
jnp	jpo	Jump if No Parity	Jump if Parity Odd
jp	jpe	Jump if Parity	Jump if Parity Even

Note: Alcuni salti condizionali hanno una forma alternativa (ad esempio JA e JNBE) ma in ogni caso si tratta di due forme diverse per descrivere la stessa condizione.

Cicli condizionali

Vi sono anche due tipi di cicli condizionali. LOOPZ è un ciclo che continua finché l'argomento rimane uguale a 0 mentre LOOPNZ continua fintantoché l'argomento rimane diverso da 0. LOOPZ ha anche un sinonimo, LOOPE ovvero "LOOP while Equal". Il sinonimo di LOOPNZ è invece LOOPNE, ovvero "LOOP while Not Equal". Ecco un semplice esempio che legge una stringa fino al primo byte nullo (uguale a 0):

```
 lea    bx, string      ; carica l'offset della stringa
 mov    cx, 100         ; la lunghezza massima della stringa è 100
loop_1:
 mov    al, [bx]        ; legge un carattere dalla memoria
 inc    bx              ; punta al carattere successivo
 cmp    al, 0           ; lo confronta con 0
 loopneloop_1           ; ripete il ciclo se non è null
 je     found
 ...                    ; nessun carattere null nei 100 byte
 ...
found:
 dec  bx                ; trovato il null, restituisce bx che punta al null
...
```

Ecco un esempio che salta tutti gli spazi iniziali di una stringa:

```
 lea    bx, string      ; carica l'offset della stringa
 mov    cx, 100         ; la lunghezza massima della stringa è 100
loop_1:
 mov    al, [bx]        ; legge un carattere dalla memoria
 inc    bx              ; punta al carattere successivo
 cmp    al, 0           ; lo confronta con 0
 je     found_null
 cmp    al, 32          ; lo confronta con lo spazio ASCII
 loope  loop_1          ; salta all'indietro se è ancora uno spazio
 jne    found
 ...                    ; nessuno spazio nei 100 byte
 ...
found:
 ...                    ; bx punta al prima carattere diverso da uno spazio
 ...
found_null:
 ...                    ; trovato il null, la stringa contiene solo spazi
 ...                    ; oppure ha lunghezza 0
```

JCXZ

JCXZ è un altro tipo di salto condizionale ma non utilizza alcun flag. Semplicemente JCXZ salta se CX è uguale a 0. Si tratta di un di metodo molto comodo per eseguire il test di CX prima dell'inizio di un ciclo se tale ciclo non deve essere eseguito quando il

valore iniziale è uguale a 0. Si ricordi: l'istruzione LOOP decrementa CX prima di verificare che sia uguale a 0. Se CX inizia dal valore 0, il ciclo verrà dunque eseguito 65.536 volte. Naturalmente, quando in CX viene inserita una costante questo test non è necessario. È invece consigliabile eseguire questo test quando in CX viene inserita una variabile.

```
   jcxz loop_skip        ; salta il ciclo se CX = 0
loop_1:
   ...
   loop  loop_1
loop_skip:
```

4.5 Manipolazione dei flag

Le nove istruzioni seguenti consentono di salvare e ripristinare i flag oppure di modificare un singolo flag.

LAHF - Load AH from Flags

LAHF copia il byte inferiore del registro dei flag nel registro AH. Questa istruzione è fornita per compatibilità con il microprocessore 8080/8085. Vengono copiati i seguenti flag: SF, ZF, AF, PF e CF.

SAHF - Store AH to Flags

LAHF copia il contenuto del registro AH nel byte inferiore del registro dei flag. Vengono copiati solo i bit 0, 2, 4, 6 e 7, poiché gli altri sono indefiniti. Tali bit corrispondono ai flag CF, AF, PF, ZF e SF. Questa istruzione è fornita per compatibilità con i microprocessori 8080/8085.

PUSHF / POPF - PUSH Flags / POP Flags

Queste istruzioni eseguono le operazioni di PUSH e POP del registro dei flag. PUSHF copia tutti i bit del registro dei flag mentre POPF non copia i bit indefiniti.

STC / CLC / CMC - SeT Carry flag / CLear Carry flag / CoMplement Carry flag

Queste istruzioni modificano direttamente lo stato del flag carry (CF).

CLD / STD - CLear Direction flag / SeT Direction flag

Queste istruzioni modificano direttamente lo stato del flag di direzione (DF) il quale controlla il fatto che le operazioni sulle stringhe incrementino o decrementino i registri di origine e di destinazione. Quando DF=1, viene eseguito un decremento.

4.6 Moltiplicazioni e divisioni

Le moltiplicazione e la divisione sono un po' più complesse rispetto a quello che si potrebbe pensare poiché quando si moltiplicano due numeri, il risultato può contenere più cifre rispetto a quelle di partenza. Quando si moltiplicano due numeri a 16 bit, il risultato può richiedere fino a 32 bit. Al contrario, quando si dividono due numeri, il dividendo (il numero che diverse essere diviso) può contenere più cifre rispetto al divisore o al quoziente. Entrambe le operazioni utilizzano per un operando il registro AL, AX o DX:AX mentre l'altro operando può essere un registro o un valore in memoria.

MUL / IMUL - MULtiply / Integer MULtiply

Queste due istruzioni eseguono la stessa operazione, tranne per il fatto che MUL è per moltiplicazioni senza segno mentre IMUL è per moltiplicazioni con segno. Uno dei numeri da moltiplicare deve essere inserito in AL o AX. L'altro operando può trovarsi in un registro o in memoria. Il registro AL o AX funge quindi da operando implicito dell'istruzione di moltiplicazione. Ad esempio:

```
mov   bl, 10
mov   al, 50
mul   bl              ; AX = AL * BL
```

Per valori più grandi, si deve usare:

```
mov   bx, 100
mov   ax, 500
mul   bx              ; DX:AX = AX * BX
```

In questo caso è richiesto l'uso di registri a 16 bit poiché almeno uno dei valori da moltiplicare è maggiore di 255. Si noti che il risultato viene memorizzato automaticamente nella destinazione implicita, ovvero in AX (8 bit × 8 bit) o in DX:AX (16 bit × 16 bit).

La moltiplicazione con segno funziona nello stesso modo. Se la metà superiore del risultato è uguale a 0, i flag carry (CF) e overflow (OF) vengono azzerati mentre in caso contrario vengono posti uguali a 1. Gli altri flag aritmetici rimangono indefiniti.

DIV / IDIV - DIVision / Integer DIVision

La divisione funziona nel seguente modo:

```
mov   ax, 500
mov   bi, 10
div   bi              ; AL = AX / DL, AH = resto
```

L'esempio precedente divide AX (16 bit) per BL (8 bit). Il risultato si trova sempre in AL e il resto in AH. Quello che segue è un esempio di divisione di un numero a 32 bit per un numero a 16 bit:

```
mov   ax, [bx]        ; carica la word bassa
mov   dx, [bx+2]      ; carica la word alta
mov   cx, [si]        ; carica il divisore
cmp   cx, 0           ; evita di eseguire la divisione per 0
je    div_by_0_err    ; (la gestione della condizione viene gestita altrove)
cmp   dx, cx          ; evita un overflow
jae   div_ovfl        ; (la gestione della condizione viene gestita altrove)
div   cx              ; AX = DX:AX/CX, DX = resto
```

Questo è un esempio tipico: il dividendo a 32 bit viene inserito in DX:AX e il divisore a 16 bit viene caricato in CX (si sarebbe anche potuto utilizzare un qualsiasi altro registro generale a 16 bit o un operando in memoria). Poiché la divisione per 0 è indefinita, nel caso si cerchi di eseguire tale divisione, l'8088 genera un interrupt 0. Quando il divisore non è una costante, è sempre consigliabile assicurarsi che sia diverso da 0. Ma questa non è l'unica possibilità di errore. Si provi a dividere 10 milioni per 10. Naturalmente la risposta è 1 milione. Il problema è che questo numero è troppo esteso per rientrare nei limiti del registro AX. Dunque si provoca un errore di overflow che genera anch'esso l'interrupt 0 così come la divisione per 0. È facile capire se si è verificato un overflow poiché questo errore si verifica quando la word alta del dividendo è maggiore o uguale al divisore. Dopo una divisione, lo stato dei flag è indefinito.

In generale l'interrupt 0 termina il programma. Si può installare un gestore dell'interrupt 0 ma in genere è più comodo verificare la possibilità di errori prima di eseguire l'operazione nel programma.

4.7 Le istruzioni BCD

Le istruzioni BCD (Binary Coded Decimal - decimali codificati in binario) operano su dati con un formato speciale. Innanzitutto vi sono due formati BCD: BCD packed e BCD unpacked. Nel primo caso vi è una cifra decimale (0-9) per nibble. Nel secondo caso vi è una cifra decimale (0-9) per byte. All'interno di un determinato intervallo di

valori, le operazioni su interi sono sempre corrette, mentre le operazioni che includono un punto decimale (ovvero le operazioni in virgola mobile) non sempre sono esatte all'interno dei limiti superiore e inferiore. La mancanza di precisione dell'aritmetica binaria spinge talvolta a utilizzare dati BCD. L'aritmetica BCD viene eseguita raggruppando le istruzioni aritmetiche intere all'interno di istruzioni BCD. Tutte le istruzioni BCD operano sui registri AL o AH. In particolare le istruzioni BCD vengono utilizzate per:

- regolare il risultato di un'operazione aritmetica su interi in modo da generare dati BCD corretti;
- regolare l'input di un'operazione aritmetica su interi per assicurare che tale operazione generi dati BCD corretti.

DAA - Decimal Adjust after Addition

L'istruzione DAA viene utilizzata dopo aver sommato due coppie di cifre BCD packed, nel modo seguente:

```
mov   al, 12h     ; carica 12h o il decimale 12 BCD packed
add   al, 39h     ; sommando 39h, il risultato binario è 4Bh
daa               ; regolazione, ora il risultato è 51h o 51 in BCD packed
```

Come si può vedere da questo esempio, DAA controlla il nibble inferiore di AL per vedere se è maggiore di 9 (in questo caso è uguale a 4Bh). In caso affermativo, al nibble inferiore vengono sottratte 10 unità e al nibble superiore viene sommata una unità. Lo stesso processo si applica per il nibble superiore. Quando si sottrae 10 dal nibble superiore, viene impostato il flag carry (CF).

DAS - Decimal Adjust after Subtraction

L'istruzione DAS si usa dopo una sottrazione di due coppie di cifre BCD packed. L'operazione è analoga alla precedente, tranne per il fatto che il flag carry viene impostato in caso di riporto.

AAA - Ascii Adjust after Addition

L'istruzione AAA si usa dopo aver sommato due cifre BCD unpacked:

```
mov   ax, 9   ; AH = 0, AL = 9
add   al, 8   ; AL = 17 (11h)
aaa           ; AH = 1, AL = 7
```

In questo esempio vengono sommati i numeri 9 e 8. Nel registro AL è contenuto il risultato corretto (17), che però non è in formato BCD unpacked. L'istruzione AAA verifica il nibble inferiore di AL per vedere se è maggiore di 9 ma il nibble inferiore è uguale a 1. A questo punto si deve considerare il flag ausiliario (AF). Durante la

somma, un bit è passato dal nibble inferiore al nibble superiore del registro AL. Se è stato impostato il flag ausiliario (AF) o se il nibble inferiore è maggiore di 9, il registro AH viene incrementato, quindi viene impostato il flag carry e viene regolato il nibble inferiore di AL. I bit del nibble alto di AL vengono sempre cancellati.

AAS - Ascii Adjust after Subtraction

L'istruzione AAS viene utilizzata dopo la sottrazione di due cifre BCD unpacked. L'operazione è simile alla precedente tranne per il fatto che quando si verifica un riporto viene decrementato il contenuto di AH e viene impostato il flag carry (CF).

AAM - Ascii Adjust after Multiplication

L'istruzione AAM viene utilizzata dopo una moltiplicazione di due cifre BCD unpacked. La moltiplicazione deve avvenire fra due numeri a 8 bit, per produrre un risultato a 16 bit:

```
mov   al, 3    ; AL = 3
mov   bl, 9    ; DL = 9
mul   b1       ; AX = 27 (1Bh)
aam            ; AH = 2, AL = 7
```

Osservando bene l'istruzione AAM ci si accorge che in realtà esegue una divisione. AAM divide AL per 10 e inserisce il risultato in AH e il resto in AL. Si noti che il risultato è invertito rispetto alla normale divisione.

AAD - Ascii Adjust before Division

L'istruzione AAD deve essere utilizzata prima di dividere cifre BCD unpacked. I registri AH e AL devono contenere una cifra BCD ciascuno e la cifra più alta deve trovarsi in AH.

```
mov   ax, 0509h     ; AH = 5, AL = 9
mov   b1, 7         ; BL = 7
aad                 ; regola AX (AX = 003Bh)
div   b1            ; AL = 59 / 7 = 8, AH = resto = 3
```

Studiando attentamente l'istruzione AAD, si scopre che in realtà converte le cifre contenute nei nibble bassi di AH e AL da un valore decimale a un valore binario o esadecimale. Il valore viene salvato nel registro AL mentre il registro AH viene cancellato.

Una caratteristica curiosa delle istruzioni AAA e AAS è il fatto che funzionano altrettanto bene su dati BCD unpacked (byte contenenti valori compresi tra 0 e 9) e su valori ASCII (corrispondenti ai caratteri ASCII da "0" a "9" o ai valori esadecimali da 30h a 39h). Questo è dovuto al fatto che il valore 3 nel nibble alto viene "logicamente" ignorato. Quando vengono sommati, i due 3 divengono un 6, cancellato dall'istruzione

AAA. In una sottrazione i due 3 diventano uno 0. Questo non si verifica nel caso di AAD e AAM. AAD richiede due cifre BCD unpacked (0-9). Altri valori generano risultati senza senso ed errori. AAM esegue una divisione per 10 di un valore binario. Pertanto può essere utilizzata per dividere AL per 10.

4.8 Istruzioni per le stringhe

Le principali istruzioni per le stringhe operano su un byte o su una word (2 byte) per volta (a partire dall'80386 operano anche su dword). Le stringhe possono essere lunghe fino a 64 KB (65536 byte pari a 32768 word). Non necessariamente le stringhe devono essere array di caratteri come avviene nei linguaggi di alto livello. Nelle istruzioni per le stringhe, i dati possono essere di qualsiasi tipo manipolabile un byte o una word per volta. In realtà si può operare su qualsiasi struttura o blocco di memoria. Le uniche cose che definiscono una stringa sono il relativo puntatore e la lunghezza. Alcune delle istruzioni per le stringhe sono precedute dal prefisso di ripetizione (REP, REPE, REPZ, REPNE o REPNZ) che consente di eseguire un'operazione su un intero blocco di memoria.

Per tutte le istruzioni sulle stringhe l'origine è sempre DS:SI e la destinazione è ES:DI. Per gli esempi di questo capitolo, si assumerà che DS ed ES facciano riferimento allo stesso segmento. Quando viene utilizzata, la lunghezza della stringa viene conservata in CX.

Uno dei vantaggi delle istruzioni sulle stringhe è il fatto che aggiornano automaticamente il registro di origine (SI) e/o il registro di destinazione (DI). Il flag di direzione determina il fatto che SI e/o DI debbano essere incrementati o decrementati dopo ogni operazione. Quando si opera su byte, ogni incremento o decremento è di 1 unità mentre per le word è di 2 unità. Tutte le istruzioni per stringhe hanno due formati assembler. Innanzitutto è possibile far seguire al codice operativo dell'istruzione la lettera B o W per specificare l'uso di byte o di word. In secondo luogo è possibile fornire operandi dichiarati altrove e l'assembler utilizzerà automaticamente il tipo di dati appropriato. È opinione dell'autore che il secondo metodo sia inutile e che convenga usare solo il primo formato. Anche quando vengono forniti gli operandi, nei registri SI e/o DI devono essere inseriti i valori corretti. Gli operandi vengono usati solo per determinare il tipo dei dati ma possono essere utili come documentazione.

Prefissi di ripetizione

I prefissi di ripetizione consentono di ripetere un'istruzione per stringhe un determinato numero di volte. Tale numero viene inserito nel registro CX che viene decrementato di 1 unità e quindi verificato dopo ogni operazione. Il decremento di CX non altera alcun flag.

REP

Il prefisso REP può essere considerato come una ripetizione che procede finché CX=0. Il prefisso REP può essere utilizzato con le istruzioni MOVS e STOS.

REPE / REPZ
I prefissi REPE e REPZ sono sinonimi dello stesso codice operativo. REPE esegue la ripetizione fintantoché gli operandi sono uguali o finché CX=0 mentre REPZ esegue la ripetizione fintantoché l'operando è uguale a 0 o finché CX=0. I prefissi REPE e REPZ possono essere utilizzati con le istruzioni CMPS e SCAS.

REPNE / REPNZ
I prefissi REPNE e REPNZ sono sinonimi dello stesso codice operativo. REPNE esegue la ripetizione fintantoché gli operandi sono diversi o finché CX=0 mentre REPNZ esegue la ripetizione fintantoché l'operando è diverso da 0 o finché CX=0. I prefissi REPNE e REPNZ possono essere utilizzati con le istruzioni CMPS e SCAS.

MOVS / MOVSB / MOVSW - MOVe String (Byte / Word)

Queste istruzioni spostano un byte o una word da DS:SI a ES:DI e incrementano o decrementano SI e DI di 1 unità (2 nel caso di MOVSW).

REP MOVS
REP MOVS sposta un blocco di byte o di word da DS:SI a ES:DI. Il risultato dell'operazione dipende dallo stato del flag di direzione (DF) e da eventuali sovrapposizioni dei blocchi di origine e di destinazione. Se DF=0 (direzione in avanti) e i blocchi non si sovrappongono, viene semplicemente eseguita una copia di byte o di word così come ci si potrebbe aspettare. Quando invece DF=1 (direzione all'indietro), l'origine (DS:SI) punta in effetti all'ultimo byte (o word) che deve essere copiato nel blocco. Inoltre, gli indirizzi che costituiscono due blocchi sovrapposti sono diversi quando DF=1.

```
lea   si, string1    ; SI = offset di string1
lea   di, string2    ; DI = offset di string2
mov   cx, 10         ; CX = 10 (lunghezza della stringa)
rep   movsb          ; string2 = string1

; dati
string1 DB 'abcdefghijk'
string2 DB 'lmnopqrstuv'
```

I dati dopo REP MOVSB:

```
string1 DB 'abcdefghijk'
string2 DB 'abcdefghijk'
```

Ecco ciò che accade quando le stringhe si sovrappongono:

```
lea   si, string1    ; SI = offset di string1
lea   di, [si+1]     ; DI = SI + 1
mov   cx, 10         ; CX = 10
rep movsb
; dati
string1 DB 'abcdefghijk'
string2 DB 'lmnopqrstuv'
```

I dati dopo lo spostamento di un byte:

```
string1 DB 'aacdefghijk'
string2 DB 'lmnopqrstuv'
```

I dati dopo lo spostamento di 10 byte:

```
string1 DB 'aaaaaaaaaaa'
string2 DB 'amnopqrstuv'
```

Al termine dello spostamento, SI e DI puntano al byte o alla word successivi e non all'ultimo oggetto spostato. Nel caso di un movimento in avanti, SI e DI punteranno al byte o alla word successivi rispetto all'ultimo spostamento. In caso contrario, SI e DI punteranno al byte o alla word precedenti rispetto all'ultimo spostamento. Nell'esempio precedente, al termine dell'istruzione REP MOVSB, SI punterà al primo byte di string2 ("a") e DI punterà al secondo byte di string2 ("m").

CMPS / CMPSB / CMPSW - CoMPare String (Byte / Word)

CMPS confronta i byte o le word in DS:SI e ES:DI e incrementa/decrementa SI e DI di 1 unità (2 nel caso di CMPSW). I codici di condizione nel registro dei flag si basano sul confronto come nel caso dell'istruzione CMP. L'incremento o decremento di SI e DI non ha alcun effetto sui flag.

REPE CMPS / REPZ CMPS

L'istruzione REPE CMPS confronta un blocco di byte o di word in DS:SI con un altro blocco in ES:DI. Viene confrontato un byte o una word per volta fintantoché rimangono uguali o fino alla fine dell'intero blocco. Al termine dell'operazione si deve verificare qual è la condizione che ne ha provocato la fine. A tale scopo si può eseguire un test del flag ZF con un salto condizionale.

```
 lea   si, string1
 lea   di, string2
 mov   cx, 10
 repe  cmpsb
 jne   no_match
match:
 ...
no_match:
 ...
string1db 'abcdefghij'
end1   db 0
string2db 'abcdefghij'
end2   db 0
```

Al termine dell'istruzione REPE CMPSB, se le due stringhe sono uguali, CX sarà uguale a 0 e SI e DI punteranno al primo byte che segue le stringhe confrontate. Nel caso precedente, SI punterà al byte END1 e DI punterà al byte END2. Il confronto richiede che i byte siano esattamente identici in quanto la CPU non considera il concetto di lettere maiuscole e minuscole. Se vengono utilizzati i dati seguenti, SI e DI e

CX avranno gli stessi valori che avrebbero dopo un confronto di due stringhe esattamente coincidenti:

```
string1db 'abcdefghij'
end1    db 0
string2db 'abcdefghiJ'   (la lettera J è maiuscola)
end2    db 0
```

L'unica operazione corretta dopo un'istruzione REP*x* CMPS*x* è un salto condizionale (anche se prima si possono salvare i flag e si possono eseguire altre istruzioni che non alterano i flag).

Quando le due stringhe non sono coincidenti, CX conterrà il numero di byte non confrontati e SI e DI punteranno ai byte che seguono gli ultimi confrontati.

REPNE CMPS / REPNZ CMPS

L'istruzione REPNE CMPS equivale a REPE CMPS tranne per il fatto che il confronto continua fino alla fine del blocco o finché non viene trovata una corrispondenza.

```
  lea    si, string1
  lea    di, string2
  mov    cx, 10
  repne  cmpsb
  jne  no_match
match:
  ...
no_match:
  ...
string1 db 'abcdefghij'
string2 db 'xxxxefghij'
```

In questo caso il confronto continua fino alle lettere "e". I registri CX, DI e SP hanno lo stesso significato già visto per l'istruzione REP CMPS.

SCAS / SCASB / SCASW - SCAn String (Byte / Word)

L'istruzione SCAS confronta il byte o la word in ES:DI con AL o AX e incrementa/decrementa DI di 1 unità (2 nel caso di SCASW). I codici di condizione nel registro dei flag vengono impostati sulla base del confronto così come avveniva nel caso dell'istruzione CMP. L'incremento o decremento di SI e DI non ha alcun effetto sui flag.

REPE SCAS / REPZ SCAS

REPE SCAS esegue la scansione della stringa in ES:DI alla ricerca di un byte o di una word che corrisponda al contenuto dei registri AL o AX. La scansione continua finché non viene trovata la corrispondenza o fino alla fine del blocco. Al termine dell'operazione si dovrebbe determinare quale condizione ha provocato la fine dell'istruzione. A tale scopo si può eseguire il test del flag ZF con un salto condizionale.

```
  lea    di, string1
  mov    al, 20h          ; inserisce lo spazio ASCII in AL
```

```
mov   cx, 10
repe  scasb
jne   no_match
match:
...
no_match:
...
string1 db '     fghij'
```

Naturalmente, in genere la scansione di una stringa non viene eseguita solo per determinare se vi è o meno una corrispondenza ma per ottenere un puntatore a una determinata locazione di memoria. Come nel caso delle istruzioni REP*x* CMPS, al termine CX contiene il numero di byte non confrontati e DI punta al byte che segue l'ultimo confrontato. In questo esempio, DI punta alla lettera "g" poiché il ciclo è terminato sulla lettera "f".

REPNE SCAS / REPNZ SCAS

L'istruzione REPNE SCAS equivale a REPE SCAS tranne per il fatto che il confronto continua fino alla fine del blocco o fino a trovare una corrispondenza. In pratica il ciclo continua fintantoché le due stringhe sono diverse.

```
lea   di, string1
mov   al, 20h        ; carica lo spazio ASCII in AL
mov   cx, 10
repne scasb
jne   no_match
match:
...
no_match:
...
string1 db '    abcde'
```

LODS / LODSB / LODSW - LOaD String (Byte / Word)

LODS legge un byte o una word da DS:SI, lo inserisce nel registro AL o AX e incrementa/decrementa SI di 1 unità (2 nel caso di LODSW). Non è necessario utilizzare un prefisso di ripetizione per LODS in quanto ogni operazione di ripetizione continuerebbe a inserire un nuovo valore in AL o in AX, cancellando il valore precedente. L'esempio seguente legge ogni byte da una stringa e converte i caratteri da minuscoli a maiuscoli:

```
lea   si, string1
mov   cx, 10
loop1:
lodsb                ; legge un byte
cmp   al, 'a'        ; salta se è prima della a minuscola
jb    next
cmp   al, 'z'        ; salta se è oltre la z minuscola
ja    next
```

```
  sub   al, 20h          ; conversione in maiuscole
  mov   [si-1],          ; AL riscritto in memoria
 next:
  loop  loop1
 string1 db 'aBcdEfGhij'
```

STOS / STOSB / STOSW - STOre String (Byte / Word)

STOS copia un byte o una word dal registro AL o AX a ES:DI e incrementa/decrementa DI di 1 unità (2 nel caso di STOSW).

```
  lea   si, string1
  lea   di, string2
 loop1:
  lodsb                  ; legge un byte
  stosb                  ; salva un byte
  cmp   al, 0            ; verifica di fine stringa
  jne   loop1
  ...
 string1 db 'abcdefghij',0 ; stringa terminata da un carattere null
 string 2 db 11 dup (0)   ; spazio per una copia di string1
```

REP STOS

L'istruzione REP STOS memorizza ripetutamente il valore di AL (o di AX) in ES:DI e incrementa/decrementa DI di 1 unità (2 nel caso delle word) dopo ogni memorizzazione.
Questa istruzione è in pratica una funzione di riempimento di un blocco. CX contiene le dimensioni del blocco in byte o in word. Questo esempio inserisce in una stringa una serie di 0.

```
 lea    di, string1
 mov    cd, 10
 cmp    al, 0
 rep    stosb
  ...
 string1 db 'abcdefghij'
```

Attenzione ai prefissi di ripetizione

Le operazioni di ripetizione sulle stringhe possono anche essere interrotte. Questo è importante poiché le operazioni su stringhe molto estese possono richiedere più di un milione di cicli! Ad esempio, su un PC IBM, il confronto di due stringhe piuttosto estese può richiedere un quarto di secondo. Ma contemporaneamente l'interrupt del timer deve acquisire il controllo circa 18.2 volte al secondo per aggiornare l'orologio interno. Quando un interrupt prende il controllo durante un'istruzione su stringhe, lo stato dell'istruzione viene automaticamente conservato dalla CPU. Tuttavia, quando

si esce dai limiti del segmento, dopo l'interrupt tale cambio di segmento viene ignorato. Questo bug è stato corretto da Intel a partire dalla CPU 80186. Se si intendono scrivere programmi che possano essere eseguiti su un 8088 o un 8086, occorre fare attenzione a non uscire dai limiti dei segmenti durante l'esecuzione esecuzione di istruzioni di ripetizione su stringhe.

4.9 Gli interrupt

Si provi a immaginare un telefono senza suoneria. Non sarebbe possibile sapere quando qualcuno sta chiamando. Probabilmente si sarebbe costretti a sollevare periodicamente il ricevitore per vedere se vi è qualcuno dall'altra parte. Ma un oggetto di questo tipo non solo sarebbe inefficiente ma diverrebbe ben presto vecchio. La suoneria del telefono funziona come un interrupt. Quando squilla il telefono in genere si interrompe quello che si stava facendo, si risponde al telefono e poi si torna alle proprie faccende. Gli interrupt funzionano esattamente nello stesso modo tranne per il fatto che un computer è veloce ed esegue sempre una sola operazione per volta. Ma prima che il computer possa rispondere a un interrupt, deve salvare un puntatore che indichi il punto al quale ritornare ed inoltre deve salvare il contenuto dei flag. Dunque quando si verifica un interrupt, la CPU esegue una PUSH del registro dei flag, del registro del segmento di codice (CS) e del puntatore dell'istruzione (IP). Quindi la CPU cambia lo stato di alcuni flag cancellando il flag trap (TF) e il flag di attivazione degli interrupt (IF) in modo da essere sicura di poter funzionare correttamente. In questo modo viene disattivata la modalità di esecuzione passo-passo (nel caso fosse in esecuzione un debugger che richieda il controllo tramite un interrupt) e vengono disattivati ulteriori interrupt finché non verrà terminata la gestione dell'interrupt in corso.

Nell'8088 vi sono 256 interrupt, numerati da 0 a 255. Ad esempio, quando si preme un tasto sulla tastiera, un segnale passa attraverso vari circuiti e raggiunge la CPU che deve interrompere ciò che stava facendo per capire ciò che è accaduto. I vari dispositivi connessi al computer (tastiera, stampanti, modem e così via) inviano segnali differenti e richiedono azioni differenti. A ogni dispositivo può essere assegnato un proprio numero di interrupt. Ad esempio alla tastiera corrisponde l'interrupt 9. Il sistema conserva un array di puntatori far che puntano alle routine di gestione di ogni interrupt, in una tabella chiamata Interrupt Vector Table (IVT). Tale array si trova all'inizio della memoria del computer, ovvero nel segmento 0, offset 0 (0000:0000). Poiché ogni puntatore far richiede 4 byte, il gestore (handler) della tastiera si trova all'indirizzo 0000:0024 (4 * 9 = 36 = 24h).

INT - software INTerrupt

L'istruzione INT genera un interrupt software. Tale interrupt viene gestito come un interrupt hardware, descritto in precedenza. Un interrupt software è simile a un'istruzione CALL far, tranne per il fatto i flag vengono inseriti (PUSH) nello stack prima di CS e IP (e naturalmente vengono cancellati i flag TF e IF).

Gli interrupt software consentono di accedere ai servizi del DOS e del BIOS. Ogni servizio ha le proprie specifiche e pertanto occorre impostare adeguatamente i registri. Tale argomento verrà affrontato più in dettaglio in seguito ma per iniziare ecco un esempio:

```
mov   ah, 9          ; 9 = funzione di visualizzazione delle stringhe
lea   dx, message    ; carica il puntatore al messaggio
int   21h            ; richiama i servizi del DOS
```

I servizi del DOS vengono richiamati tramite l'interrupt 21h. Si tratta di un numero scelto arbitrariamente e senza alcun significato particolare. Ad esempio la funzione 9 dei servizi DOS è una richiesta di visualizzazione di una stringa.

IRET - Interrupt RETurn

L'istruzione IRET consente di uscire da un interrupt, un po' come avviene per l'istruzione RET per le procedure tranne il fatto viene eseguita la POP dei flag dallo stack. È necessario utilizzare questa istruzione solo se si scrivono routine di servizio per la gestione degli interrupt. IRET funziona nel seguente modo:

```
pop   ip             ; non è possibile eseguire queste operazioni
pop   cs             ; che però vengono eseguite da IRET
popf
```

CLI - CLear Interrupt flag

L'istruzione CLI cancella il flag di attivazione degli interrupt ovvero disabilita gli interrupt. L'istruzione disabilita solo gli interrupt esterni e non gli interrupt non mascherabili (NMI). Dopo l'esecuzione dell'istruzione CLI, sarà dunque consentito l'ingresso solo degli interrupt NMI. CLI deve essere utilizzata durante l'esecuzione di punti critici del codice che non possono essere interrotti. Al termine di questa porzione di codice critica, si possono riattivare gli interrupt tramite l'istruzione STI.

Nell'8088 vi sono due tipi di interrupt hardware. Il primo tipo chiede di eseguire una determinata azione (è questo il caso degli interrupt della tastiera e dei timer). Tali interrupt possono essere disabilitati o mascherati. È un po' come spegnere la suoneria del telefono quando non si vuole essere disturbati. Il secondo tipo è costituito dagli interrupt non mascherabili (NMI), ovvero dell'interrupt 2. Non è mai possibile disattivare questi interrupt. In pratica corrispondono a una "chiamata d'emergenza" che non può essere ignorata. Tali interrupt vengono ad esempio impiegati dai server di rete.

STI - SeT Interrupt flag

L'istruzione STI attiva il flag di attivazione degli interrupt ovvero riattiva gli interrupt. Tale istruzione può essere utilizzata dopo una porzione critica di codice preceduta da

CLI oppure in una routine di gestione degli interrupt nel momento in cui è possibile accettare l'arrivo di ulteriori interrupt.

4.10 Istruzioni varie

XCHG

L'istruzione XCHG scambia il contenuto di due registri o di un registro e di una locazione di memoria:

```
xchg    ax, bx          ; temp = AX, AX = BX, BX = temp
```

XLAT

L'istruzione XLAT esegue una traduzione di un byte in AL ricercando la traduzione in una tabella. L'indirizzo della tabella corrisponde al contenuto del registro BX. XLATB e XLAT sono la stessa istruzione.

```
mov     bx, OFFSET xlat_tbl
mov     al, 5
xlat                    ; AL = [BX+AL]
```

LEA - Load Effective Address

L'istruzione LEA carica l'indirizzo effettivo dell'operando di origine nell'operando di destinazione. In altre parole nella destinazione viene copiato l'offset dell'operando di origine (e non il suo valore). La destinazione è sempre un registro. L'origine può essere una qualsiasi espressione di indirizzamento della memoria. Se l'origine contiene solo uno scostamento, la maggior parte degli assembler può utilizzare automaticamente il formato più efficiente: MOV reg OFFSET mem.

```
lea     bx, [si]                ; non è utile poiché
mov     bx, si                  ; equivale a questa
lea     bx, data_item           ; queste sono uguali
mov     bx, OFFSET data_item
```

L'esempio successivo mostra l'ottimizzazione di un programma tramite l'impiego dell'istruzione LEA. Ad esempio, LEA può sommare due registri e una costante.

```
lea     di, [si+bx+4)   ; DI = SI + BX + 4
mov     di, si          ; queste tre istruzioni
add     di, bx          ; eseguono la stessa operazione di LEA
add     di, 4
```

LDS / LES - Load far pointer using DS / ES

L'istruzione di caricamento di un puntatore far trasferisce un segmento e un offset dalla memoria in un registro di segmento e in un qualsiasi registro a 16 bit (AX, BX, CX, DX, DI, SI, BP e SP). Il puntatore far deve essere memorizzato partendo dall'offset seguito dal segmento. Ad esempio:

```
les     di, str_tr2             ; ES = segmento contenente str2
                                ; DI = offset di str2
lds     si str_ptr1             ; DS = SEG str1, SI = OFFSET str1
; questi dati sono dichiarati altrove:

str_ptr1 dd str1
str_ptr2 dd str2
str1 db 'string1'
str2 db 10 dup(0)
```

CBW - Convert Byte to Word

CBW converte un byte con segno contenuto in AL in una word in AX mantenendone il segno. L'operazione viene eseguita copiando il bit di segno (il bit 7) di AL in ogni bit di AH.

CWD

CWD converte una word con segno in AX in una dword in DX:AX, mantenendone il segno. L'operazione viene eseguita copiando il bit di segno (il bit 15) di AX in ogni bit di DX

NOP

L'istruzione NOP non esegue nulla. Tale istruzione può essere utilizzata per cancellare altre istruzioni o per riservare spazio durante il debugging. Alcune volte queste istruzioni vengono inserite dagli assembler quando tendono a riservare il massimo spazio che può essere richiesto dalle istruzioni. Inoltre le istruzioni NOP possono essere utilizzate per allineare l'inizio di cicli sui limiti di una word o di una dword.

4.11 Riepilogo dei flag

Vi è un ordine logico nel modo in cui le varie istruzioni modificano lo stato dei flag. In generale, quando si trasferiscono dati non viene alterato alcun flag mentre quando si confrontano o modificano i dati con operazioni aritmetiche i flag vengono modificati. La Tabella 4.5 contiene un riepilogo di tutte le operazioni che modificano i flag.

Tabella 4.5 Istruzioni che modificano i flag aritmetici.

	CF	PF	AF	ZF	SF	OF
add/sub	M	M	M	M	M	M
adc/sbb	TM	M	M	M	M	M
cmp/cmps/scas	M	M	M	M	M	M
inc/dec	M	M	M	M	M	M
and/or/xor/test	0	M	-	M	M	-
shift	M	M	-	M	M	M
rotate	M	M				
aaa/aas	M	-	M	-	-	-
aad/aam	-	M	-	M	M	M
daa/das	T			M	M	M
mul/imul	M	-	-	-	-	M
div/idiv	-	-	-	-	-	-
stc	1					
clc	0					
cmc	TM					
popf/sahf	M	M	M	M	M	M

CF = carry; PF = parità; AF = ausiliario; ZF = zero; SF = segno; PF = overflow

Le istruzioni che non modificano i flag sono: MOV, LEA, PUSH, PUSHF, POP, NOT, XCHG, CALL, RET, JMP, JCC, LOOP, LOOPxx, LODS, STOS, MOVS, HLT, IN, OUT, CBW, CWD, LAHF, LDS, LES e LOCK

Le istruzioni che modificano flag diversi da quelli aritmetici sono: CLD, STD, CLI, STI e IRET.

Nota: M = modifica in 0 o 1; T = test del flag; 1 = assegna 1; 0 = assegna 0; - = non definito; nulla = non modificato.

Capitolo 5

Il primo programma

Nel capitolo precedente si è parlato delle istruzioni di base che l'8088 è in grado di eseguire. In questo capitolo tali istruzioni verranno utilizzate per realizzare un programma completo. Questo significa che si dovrà parlare della struttura dei programmi, delle regole per la creazione delle etichette e dell'organizzazione in memoria dei programmi.

5.1 Direttive assembler

L'esempio seguente è forse il più piccolo programma assembler in grado di eseguire qualcosa di visibile. Come si noterà, nel programma vi è una serie di istruzioni della CPU e di direttive assembler. Si è già parlato delle istruzioni della CPU e dunque è il momento di introdurre il concetto di direttiva assembler. Le direttive assembler non sono istruzioni che devono essere eseguite dalla CPU ma piuttosto indicazioni o suggerimenti per l'assembler per consentirgli di tradurre correttamente le istruzioni della CPU, per controllare il listato e per varie altre attività. Le direttive sono anche chiamate pseudo-operazioni.

```
model small
.stack
.code
main proc
mov ax,@data
mov ds,ax
lea dx,msg
mov cx,7
mov bx,1
mov ah,40h
int 21h
mov ah,4ch
```

```
; Un completo programma di esempio
;-------------------------------------------
; Salve.asm Un cordiale saluto
;
; Semplice programma che visualizza un messaggio
;-------------------------------------------
.model small
.stack
.code
main proc
  mov  ax, @data        ; carica il segmento dati
  mov  ds, ax
  lea  dx, msg          ; carica l'indirizzo del messaggio
  mov  cx, 7            ; lunghezza di msg
  mov  bx, 1            ; stdout
  mov  ah, 40h          ; funzione DOS di scrittura
  int  21h              ; chiamata al DOS
  mov  ah, 4ch          ; funzione DOS di uscita
  int  21h              ; chiamata al DOS
main endp
.data
msg db 'Salve' ,13,10
```

Figura 5.1 Il programma di esempio.

```
int 21h
main endp
.data msg
db 'Ciao!',13,10
end main
```

Anche se questo esempio funziona, vi sono alcuni problemi. Il formato del codice non è molto leggibile e non vi sono commenti che indichino lo scopo del programma o il motivo per il quale si è scelta una determinata istruzione.

Questa versione del programma è molto più chiara. In particolare si noti la presenza dei commenti (preceduti da un punto e virgola). In genere la formattazione e la presenza di commenti più o meno abbondanti nel codice sono legati allo stile personale. Dunque si può scegliere lo stile che si trova più comodo e appropriato per il proprio livello di esperienza e per coloro che si troveranno a eseguire la manutenzione del codice.

Cosa significano tutte queste istruzioni?

La maggior parte degli assembler è dotata di una serie di direttive specifiche. Ad esempio l'assembler Microsoft MASM 6.11 ne ha un centinaio. Per apprenderne l'uso si può fare riferimento alla documentazione dell'assembler. Lo scopo di questo manuale è quello di mostrare l'utilizzo ottimale delle istruzioni della CPU e dunque si parlerà delle sole direttive necessarie per scrivere un programma utile e completo.

La direttiva ".model" è stata introdotta a partire dal MASM 5.0 (ma si trova in tutte le versioni di TASM). Questa direttiva, insieme a .stack, .code e .data, forma il gruppo delle direttive semplificate di segmentazione. Prima di queste direttive, era necessario dichiarare i segmenti del codice, dei dati e dello stack in modo molto più complesso e soggetto a errori; tale metodo verrà discusso nel Capitolo 18. Esistono dei vantaggi nell'utilizzo del vecchio metodo e questo è il motivo per il quale è consigliabile apprenderli entrambi. Inoltre in un file è possibile utilizzare insieme i due metodi.

Per utilizzare le direttive semplificate di segmentazione, si deve utilizzare la direttiva .model prima di ogni altra istruzione che generi codice o dati. Tale direttiva dichiara un modello di memoria (vedere la Tabella 5.1) per il programma. A questo punto basterà utilizzare la direttiva .code prima di scrivere il codice e la direttiva .data prima di dichiarare i dati. Dunque è tutto molto semplice.

A questo punto ci si potrebbe chiedere "Quali sono i modelli di memoria e cosa è necessario sapere a proposito?". La risposta è che il concetto di modello di memoria è dovuto alla natura segmentata dell'architettura di memoria dei microprocessori 80x86. Se non vi fossero segmenti, non vi sarebbe la necessità di utilizzare modelli di memoria. Più avanti si parlerà più in dettaglio dei vari modelli di memoria disponibili.

5.2 Etichette e identificatori

Gli identificatori sono nomi (o simboli) che il programmatore può inventare per definire variabili, costanti, segmenti, procedure, etichette del codice ed elementi di un programma. Le etichette sono identificatori che definiscono gli indirizzi all'interno di un programma. Tali etichette possono essere definite in vari modi, ma normalmente si specifica il carattere ":" dopo il nome dell'etichetta. Le etichette possono trovarsi in una propria riga di codice o possono essere seguite da istruzioni per la CPU. In entrambi i casi, il loro indirizzo è quello del primo byte dell'istruzione successiva. Le etichette sono utilizzate come indirizzo di destinazione per salti e chiamate. Il carattere ":" inserito dopo il nome dell'etichetta deve essere specificato solo quando si definisce l'etichetta e non quando si fa riferimento ad essa per un salto. Non è possibile definire un'etichetta più di una volta anche se è ovviamente possibile fare riferimento ad essa più volte.

Nell'esempio precedente è stata definita l'etichetta FINISH alla quale non fa però riferimento alcuna istruzione. Si possono definire e utilizzare le etichette nel modo desiderato. In questo esempio vi sono tre etichette:

```
 call  get_a_key          ; restituisce il codice ASCII in AL
 cmp   al, 'A'            ; confronto con la A maiuscola
 jb    done               ; se è inferiore, salta
 cmp   al, 'Z'            ; confronto con la Z maiuscola
 ja    not_upper          ; se è superiore, salta
 add   al, 32             ; conversione in minuscole
 jap   done
not_upper:
 cmp   al, 'a'            ; confronto con la a minuscola
```

```
jb     done            ; se è inferiore, salta
cap    al, 'z'         ; confronto con la z minuscola
ja     done            ; se è superiore, salta
sub    al, 32          ; conversione in maiuscole
done:
```

Due delle etichette ("not_upper" e "done") sono definite e utilizzate; l'etichetta "done" viene utilizzata in quattro punti. L'etichetta "get_a_key" viene utilizzata ma non è definita. In un programma completo, tale etichetta dovrebbe essere definita in qualche altro punto del programma ma può esistere una sola definizione di un'etichetta.

I nomi utilizzabili come identificatori sono soggetti a numerose restrizioni. In particolare gli identificatori possono iniziare con uno dei seguenti caratteri:

A-Z
a-z
$ (carattere di dollaro)
% (carattere di percentuale)
. (carattere punto)
? (punto interrogativo)
@ (simbolo "at")
: (carattere di sottolineatura)

Il secondo carattere deve contenere un carattere dello stesso tipo (tranne il punto) ma può anche contenere le cifre da 0 a 9. In genere si cerca di evitare l'uso del carattere "at" (@) poiché è utilizzato da molti simboli interni dell'assembler. Inoltre, per chiarezza, si dovrebbe evitare di utilizzare il punto.

Infine non è possibile utilizzare identificatori (o simboli) aventi lo stesso nome di una direttiva assembler o di un'istruzione della CPU. Nei simboli non viene fatta alcuna distinzione fra lettere maiuscole e minuscole e in generale tutti i simboli vengono internamente convertiti dall'assembler in lettere maiuscole (questo comportamento può essere modificato tramite le opzioni dell'assembler). I simboli possono essere lunghi fino a 31 caratteri. Alcuni assembler consentono di utilizzare simboli più estesi ma considerano significativi solo i primi 31 caratteri.

Procedure

La direttiva PROC consente di definire un'etichetta che identifica l'inizio di una procedura o di una funzione. Per concludere la procedura si deve utilizzare lo stesso identificatore seguito da ENDP.

@DATA

@DATA è un simbolo predefinito che restituisce il nome del segmento dati quando viene impiegato uno schema semplificato di assegnamento del nome dei segmenti.

Definizione dei dati

I dati vengono definiti e dichiarati utilizzando una o più direttive per la definizione dei dati. Le direttive più utilizzate sono:

DB definisce un byte;
DW definisce una word;
DD definisce una dword.

La direttiva può essere seguita da uno o più oggetti. Opzionalmente è possibile assegnare un nome al primo oggetto della riga. Ecco alcuni esempi:

```
db     1
db     2
db     'a'
db     "non"
dw     1,2,3
dd     4
name1  db     'un nome'
```

5.3 Uso delle funzioni di sistema del DOS

Il programma d'esempio SALVE utilizza due funzioni DOS. L'accesso alle funzioni di sistema del DOS e del BIOS avviene tramite interrupt software. È disponibile un gran numero di interrupt di sistema tanto che anche una descrizione sommaria dei più importanti richiederebbe un intero libro. In realtà anche un semplice elenco degli interrupt di sistema disponibili occuperebbe un libro, pertanto per informazioni approfondite si rimanda a pubblicazioni specifiche.

Il programma SALVE utilizza le due funzioni DOS interrupt 21h, funzione 40h (scrivi su un file o su un dispositivo) e interrupt 21h, funzione 4Ch (esci dal programma). Prima il numero della funzione viene inserito nel registro AH e quindi viene richiamato il DOS utilizzando l'interrupt 21h. La documentazione relativa agli interrupt contiene i registri che devono essere predisposti e i valori restituiti.

5.4 La direttiva END

Ogni file deve includere una direttiva END che deve trovarsi alla fine del file. La direttiva END può anche contenere un'etichetta che verrà utilizzata come indirizzo iniziale del programma. Quando si esegue il link di più file, solo un file può contenere una direttiva END con l'indirizzo iniziale.

La Figura 5.3 mostra un modello generico per un programma utilizzabile nel modello di memoria small.

```
Funzione 40h:

input:
   AH          40h
   BX          handle del file (1 per lo schermo)
   CX          lunghezza del blocco da scrivere
   DS:DX       puntatore al blocco da scrivere
output:
   CF          =1 in caso di errore
   AX          codice d'errore se CX=1 oppure numero dei byte scritti


Funzione 4Ch:

input:
   AH          4Ch
   AL          codice di uscita del programma (usato da ERRORLEVEL nei file batch)
output:
   non torna al programma
```

Figura 5.2

```
;-------------------------------------------
; Modello di programma ASM  generico
;-------------------------------------------
. model small
.stack
.code
<nome> proc
   mov  ax, @data          ; carica il segmento dati
   mov  ds, ax
   ; < codice >
   mov  ah, 4ch            ; funzione DOS di uscita
   int  21h                ; chiamata al DOS
<nome> endp
. data
 ; < dati >
end <nome>
```

Figura 5.3

5.5 I modelli di memoria

La maggior parte degli esempi di questo manuale utilizza il modello di memoria small. I modelli di memoria sono convenzioni stabilite da Microsoft e dall'industria del software. I programmi assembler possono contenere un misto di vari modelli di memoria. In generale in un programma si deve però scegliere un modello di memoria e utilizzare sempre quello, a meno che si sappia esattamente ciò che si sta facendo.

Quando si richiama una procedura (o si salta a un'etichetta), la nuova procedura si può trovare nello stesso segmento di codice o in un segmento diverso. Una chiamata a una procedura che si trova in un altro segmento è una chiamata far mentre una chiamata all'interno del segmento è una chiamata near. È anche possibile eseguire un salto incondizionato (JMP) a un indirizzo near o far. Un'analogia potrebbe essere quella delle chiamate telefoniche urbane e in teleselezione. E così come una chiamata in teleselezione costa più di una chiamata urbana, anche una chiamata far ha un suo costo in termini di dimensioni del codice e di cicli di CPU.

Se tutto il codice del programma rientra in un segmento, tutte le chiamate e i salti saranno di tipo near. Ma quando un programma occupa più di un segmento, sorge qualche problema. La soluzione più semplice consiste nel convertire ogni chiamata e ogni rientro in istruzioni far. Se l'unico problema fosse legato al codice, vi sarebbero solo due modelli di memoria. Ma occorre anche considerare il numero dei segmenti contenenti i dati. Quando a una procedura si passa uno o più puntatori a dati, si deve specificare anche il segmento che contiene tali dati.

Tutti i puntatori ai dati sono puntatori far (ovvero sono costituiti da un segmento e da un offset). Gli unici problemi sono legati al numero dei puntatori che condividono lo stesso registro di segmento e al fatto che più registri di segmento siano uguali. Dunque parlando di segmenti per i dati occorre sapere se i dati del programma possono rientrare in un segmento (64 KB) o se è necessario utilizzare due o più segmenti (vedere la Tabella 5.1).

Programmando in modalità protetta a 32 bit, possono essere utilizzati questi stessi modelli di memoria ma le dimensioni di un segmento saranno di 4 GB invece di 64 KB. Poiché 4 GB è il limite dello spazio di indirizzamento, non ha più senso scrivere un programma che utilizzi più segmenti per i dati o per il codice anche se talvolta vi sono validi motivi per comportarsi in questo modo.

Vi è anche un nuovo modello di memoria chiamato flat. Questo è l'equivalente a 32 bit del modello tiny. Tale modello rende la programmazione per i microprocessori dall'80386 in avanti uguale a quella di qualsiasi altro microprocessore a 32 bit (come un DEC VAX, una Sun Sparc o un Motorola 680x0). Semplicemente il sistema operativo imposta tutti i registri di segmento allo stesso valore e dunque il programma non deve più preoccuparsene.

L'unico problema del modello di memoria flat è il fatto che aumenta la semplicità a scapito di alcuni dei vantaggi della modalità protetta, ovvero la protezione. In un modello small a 32 bit il codice è protetto contro modifiche accidentali ed è impossibile scrivere nel segmento dello stack quando si accede ai dati, dunque vi è una maggiore protezione contro i blocchi di sistema. Il modello di memoria flat non fornisce questa protezione.

Tabella 5.1 Modelli di memoria.

Modello	Segmenti di codice	Segmenti per i dati	Note
tiny	1	1	CS=DS=ES=SS
small	1	1	ES=DS
compact	1	>1	più segmenti per i dati
medium	>1	1	ES=DS, più segmenti per il codice
large	>1	>1	più segmenti sia per il codice che per i dati
huge	>1	>1	un singolo array può essere >64 KB

Dopo aver visto cosa è necessario utilizzare per scrivere programmi per l'8088, è ora di occuparsi dei set di istruzioni dei microprocessori 186, 286, 386, 486 e Pentium per poi affrontare l'argomento dell'ottimizzazione dei programmi per il Pentium.

Capitolo 6

Strumenti assembler

Gli strumenti si comportano meglio quando vengono utilizzati per gli scopi per i quali sono stati creati. Dunque è necessario sapere quali strumenti sono dedicati a eseguire una determinata operazione. Si potrebbe avere in mano uno strumento molto potente ma utilizzarlo per un'operazione per la quale non è stato progettato. D'altra parte, talvolta è necessario eseguire un'operazione potendo contare solo sugli strumenti che si hanno a disposizione. Può capitare molto spesso di utilizzare lo strumento errato. Normalmente ci si accorge di non avere fra le mani lo strumento ottimale mentre altre volte non si sa quale strumento utilizzare.

6.1 Editing

Dopo la fase progettuale si inizia a scrivere il programma utilizzando un editor di testi o un programma di videoscrittura. Normalmente si preferisce utilizzare un editor di testi che si rivela molto più comodo per scrivere programmi (al contrario dei programmi di videoscrittura che sono in genere più adatti per scrivere documenti). Un'altra possibilità consiste nell'impiegare l'editor fornito dall'ambiente di sviluppo integrato, ovvero l'IDE Borland o il PWB Microsoft. Presto o tardi tutti impiegano qualche tipo di ambiente integrato ma anche in questo caso è necessario conoscere tutti i singoli passi necessari per creare un programma eseguibile.

Ogni programmatore sembra avere un proprio editor preferito ed è estremamente difficile cambiare editor dopo essersi abituati a un determinato prodotto.

6.2 Assemblaggio

Tramite l'editor il programmatore crea uno o più file di codice sorgente assembler e in alcuni casi file di codice sorgente in altri linguaggi. I file assembler dovrebbero avere

l'estensione .ASM. Tali file possono poi essere assemblati utilizzando un assembler (MASM o TASM) nel seguente modo:

```
C:> masm esempio;
  Microsoft (R) Macro Assembler Version 5.10
  Copyright (C) Microsoft Corp 1981, 1988. All rights reserved.
  49696 + 129629 Bytes symbol space free
    0 Warning Errors
    0 Severe Errors

C:> tasm esempio;
  Turbo Assembler Version 2.02
  Copyright (c) 1988, 1990 Borland International
  Assembling file:    esempio.ASM
  Error messages:     None
  Warning messages:   None
  Passes:       1
  Remaining memory:   167k
```

L'assembler crea un file .OBJ che corrisponde alla traduzione in codice macchina del programma. Se il codice sorgente contiene errori, l'assembler non genererà alcun file .OBJ. Al suo posto produrrà un elenco di errori che devono essere corretti. Gli errori segnalati non sono errori di progetto o logici ma istruzioni che l'assembler non è in grado di tradurre in codice macchina.

Gli esempi presentati in questo manuale richiedono l'uso dell'assembler MASM 5.0 o successivo o una qualsiasi versione dell'assemblerTASM. A partire dall'assembler MASM 6.0 Microsoft ha notevolmente cambiato il prodotto. In particolare il programma assembler ML.EXE richiama anche il linker, così come avviene nel caso del compilatore C (CL.EXE). Inoltre è stato creato un nuovo programma MASM.EXE che accetta le stesse opzioni del MASM 5.1, le converte nel formato ML e quindi richiama il programma ML.EXE. Per i dettagli si consiglia di consultare la documentazione dell'assembler utilizzato. Si può utilizzare il metodo preferito o il metodo richiesto dai programmi; in ogni caso, in questo manuale si utilizzerà il termine MASM per far riferimento a tutte le versioni dell'assembler Microsoft.

6.3 **Linking**

Dopo aver corretto tutti gli errori presentati dall'assembler, si deve utilizzare un linker per riunire uno o più file.OBJ e creare un file eseguibile .EXE. Il linker (LINK nel caso di Microsoft o TLINK nel caso di Borland) collega tutti i riferimenti ad etichette presenti in un file alle relative definizioni contenute in altri file. I seguenti esempi mostrano il link di un singolo file utilizzando i programmi LINK e TLINK:

```
C:> link esempio;
Microsoft (R) Segmented Executable Linker Version 5.30
Copyright (C) Microsoft Corp 1984-1992. All rights reserved.
```

```
C:> tlink esempio
Turbo Link Version 3.01 Copyright (c) 1987, 1990 Borland International
```

6.4 Debugging

Dopo che la fase di link del programma è stata completata con successo, si deve eseguire il test del file eseguibile. Per utilizzare un debugger come CodeView di Microsoft o Turbo Debugger di Borland è consigliabile includere nel file .EXE alcune informazioni di debug. A tale scopo si devono selezionare le opportune opzioni dell'assembler in modo da generare informazioni sui simboli e i numeri di riga nei file .OBJ. Si potrà così utilizzare un'opzione del linker che raccoglie queste informazioni e le aggiunge alla fine del file .EXE. Quando il DOS (o un qualsiasi altro sistema operativo) carica il file .EXE, ignora tutte le informazioni di debug ma quando il file .EXE viene caricato da un debugger, tale programma utilizzerà le informazioni di debug inserendole in una tabella e quindi eseguirà la parte di codice del file .EXE. Ad esempio, nel caso del debugger CodeView Microsoft si devono utilizzare i seguenti comandi:

```
masm/zi esempio;
link/co esempio;
cv esempio.exe
```

L'opzione /zi genera le informazioni per CodeView e le memorizza nel file .OBJ. L'opzione /co inserisce le informazioni per CodeView nel file .EXE. Quindi viene eseguito il debugger CodeView.

Nel caso del Turbo Debugger Borland, i comandi da utilizzare sono:

```
tasm/zi esempio;
tlink/v esempio;
td esempio.exe
```

DEBUG32

DEBUG32 è un debugger a 32 bit contenuto nel disco fornito con questo manuale. DEBUG32 è abbastanza simile al programma DEBUG del DOS ma fornisce una serie di miglioramenti e molte funzionalità avanzate come ad esempio:

- il supporto dei registri e dell'indirizzamento a 32 bit;
- il debugging in modalità protetta;
- il supporto di applicazioni DPMI;
- il supporto della memoria EMS.

DEBUG32 è un debugger a riga di comando, proprio come DEBUG ma è disponibile una versione commerciale di questo programma dotata di finestre, visualizzazione del codice sorgente ed altre interessanti funzionalità. Questo programma è stato cre-

ato da Rob Larson della Larson Computing. Debugger di questo tipo sono necessari per eseguire il debug di codice a 32 bit. Per scrivere vero codice a 32 bit si deve eseguire il programma in modalità protetta in un segmento che sia stato specificato per codice a 32 bit. Come si vedrà nel Capitolo 18, i programmi operanti in modalità protetta possono avere codice che rientra tutto in segmenti a 16 bit (chiamati USE16) e/o in segmenti a 32 bit (chiamati USE32).

La sigla DPMI, DOS Protected Mode Interface, è una specifica che consente ai programmi DOS di accedere alle funzionalità avanzate presenti a partire dal microprocessore 80386 utilizzando uno stile corretto che non compromette la stabilità del sistema. Con DPMI si intende una serie di funzioni che gestiscono la memoria di sistema, le modalità di commutazione, gli interrupt e le comunicazioni con programmi operanti in modalità reale. Gli ambienti multitasking operanti in modalità protetta, i gestori di memoria e i sistemi operativi che implementano funzioni DPMI sono chiamati host DPMI. Le applicazioni operanti in modalità protetta che richiedono servizi a un host DPMI sono chiamati client DPMI. Alcuni esempi di host DPMI sono:

- Windows 3.0 e successivi;
- 386MAX (Qualitas)
- QEMM (Quarterdeck)
- NETROOM e Cloaking Developers' Toolkit (Helix)
- DOS 7 (Novell)

Nel Capitolo 18, si proverà a scrivere un'applicazione a 32 bit operante in modalità protetta che funge da client DPMI e che può essere eseguita sotto DOS. Il debugging di un programma client DPMI richiede l'uso del debugger DEBUG32. Infatti, se viene eseguita una chiamata a una funzione DPMI per passare dalla modalità reale alla modalità protetta, i debugger CodeView e Turbo Debugger bloccano il sistema.

Ulteriori informazioni e un elenco completo dei comandi disponibili in DEBUG32 si trovano nell'Appendice H.

Quando si scrivono programmi si possono utilizzare vari tipi di strumenti. Le categorie presentate in questo capitolo sono il minimo indispensabile per lo sviluppo di programmi. Se necessario si devono però utilizzare anche altri strumenti come le librerie, i generatori di codice, i sistemi di controllo delle versioni, i gestori di test e così via. Gli esempi presentati in questo manuale sono relativamente semplici in termini di dimensioni globali del progetto e non richiedono nessuno di questi strumenti avanzati. Nei Capitoli 11 e 12 verranno introdotti altri due strumenti forniti nel disco fornito con questo manuale.

Capitolo 7

Evoluzione del set di istruzioni: dal 186 al 386

In questo capitolo, verranno discusse le modifiche più importanti apportate al set di istruzioni e all'architettura dell'8086 nell'evoluzione verso l'80386. Ogni nuovo chip poteva funzionare a frequenze più elevate e portava con sé alcune nuove istruzioni.

Le modifiche possono essere riassunte dal seguente schema:

- Il 186 ha aggiunto nuove istruzioni e nuove forme di istruzioni utili per la programmazione di applicazioni.
- Il 286 ha aggiunto la modalità protetta e le istruzioni necessarie per controllare i programmi che sfruttavano tale modalità.
- Il 386 ha aggiunto alcune nuove istruzioni, nuovi formati e codice a 32 bit, modalità di indirizzamento avanzate, la modalità protetta a 32 bit e le funzioni di memoria virtuale paginata.

7.1 L'80186

Il 186 è il meno conosciuto della famiglia 80x86. Tale chip è stato progettato per l'utilizzo nei circuiti interni dei sistemi e dunque non è mai apparso nel mercato dei PC. La maggior parte delle modifiche apportate al set di istruzioni che normalmente sono attribuite al 286 sono in realtà state introdotte nel 186.

BOUND

L'istruzione BOUND verifica un registro per determinare se si trova all'interno dei limiti di un array. Se tale condizione non è vera, viene generato l'interrupt 5. Proprio per quest'ultimo fatto, tale istruzione viene utilizzata con una certa rarità. L'operando in memoria è sempre composto da due word che formano il limite inferiore seguito dal limite superiore.

```
bound  reg16, mem32
```

ENTER e LEAVE

Le istruzioni ENTER e LEAVE possono essere utilizzate per impostare uno stack frame, ovvero la parte di informazioni generata sullo stack all'ingresso di una procedura per il passaggio dei parametri e la memorizzazione delle variabili locali, un'operazione molto comune nella maggior parte dei linguaggi di alto livello. ENTER crea uno stack frame all'inizio di una procedura e LEAVE è l'istruzione complementare per l'uscita dalla procedura.

```
enter   imm16, level
```

L'operando imm16 conta il numero dei byte da riservare per le variabili locali. level è uguale a 0 per le convenzioni di chiamata utilizzate dalla maggior parte dei linguaggi, inclusi il C, il BASIC e il FORTRAN (in Pascal l'operando level consente a una procedura di accedere alle variabili locali delle procedure chiamanti ma in questo manuale verrà utilizzato solo il livello 0). Il seguente esempio mostra la struttura di impostazione di uno stack frame:

```
enter   4, 0    ; crea uno stack frame di 4 byte
                ; nella memoria locale
nop             ; non fa nulla
leave           ; chiude lo stack frame
ret             ; uscita
```

che equivale a:

```
push bp         ; salva BP
mov  bp, sp     ; imposta in BP lo stack frame
sub  sp, 4      ; crea lo spazio per 4 byte
nop             ; non fa nulla
mov  sp, bp     ; chiude lo stack frame
pop  bp         ; ripristina BP
ret             ; uscita
```

INS e OUTS

INS e OUTS sono istruzioni ripetitive per le stringhe che eseguono l'input e l'output come le istruzioni IN e OUT. Nel caso di INS la destinazione è specificata in ES:DI, la porta di input in DX e il numero di elementi da leggere in CX. INSB riceve i dati in forma di byte mentre INSW li riceve in forma di word. DI viene modificato dopo la lettura di ogni byte o word sulla base del flag di direzione. OUTS esegue l'operazione opposta in cui DS:SI punta alla stringa di dati o di word da inviare; DX è la porta di output e CX corrisponde al numero di elementi da scrivere. In ogni caso si tratta di istruzioni utilizzate raramente.

IMUL (moltiplicazione con segno)

A partire dal 186 sono disponibili due nuove forme dell'istruzione IMUL. Si tratta di forme a due e tre operandi. Nella forma a due operandi, uno dei fattori nonché la destinazione è costituita da un registro a 16 bit. L'altro fattore è dato da una costante. Nella forma a tre operandi, la destinazione corrisponde al primo operando che deve essere un registro a 16 bit mentre i due fattori successivi sono un registro a 16 bit o un

operando in memoria seguiti da una costante. Se il risultato è troppo esteso, vengono impostati i flag di overflow e carry.

```
imul   reg16, immed
imul   reg16, reg16, immed
imul   reg16, mem16, immed
```

PUSH imm

L'istruzione PUSH del 186 accetta un operando costituito da un valore immediato (costante). Tale operazione viene utilizzata principalmente per passare una costante come parametro di una procedura:

```
push   6
call   print_num
```

PUSHA e POPA

Le istruzioni PUSHA (PUSH All) e POPA (POP All) inseriscono o estraggono dallo stack un numero di registri secondo un ordine ben preciso. Per PUSHA l'ordine è: AX, CX, DX, BX, SP, BP, SI, DI. Il valore di SP è quello che precedeva l'esecuzione dell'istruzione. Per POPA si utilizza l'ordine inverso: DI, SI, BP, SP, BX, DX, CX, AX. Il valore di SP non viene considerato da POPA.

Rotazioni e scorrimenti

Tutte le istruzioni di rotazione e di scorrimento del 186 e dei microprocessori successivi accettano un valore immediato. Precedentemente l'unico valore costante specificabile era 1. Se il valore immediato è maggiore di 31, viene troncato a 31. Le istruzioni interessate da questa modifica sono: RCL, RCR, ROL, ROR, SHL, SHR, SAL e SAR.

7.2 L'80286

Le modifiche incluse a partire dal 286 comprendono le operazioni in modalità protetta, nuove istruzioni e cicli più veloci. La modalità protetta è, fra le nuove delle funzionalità, una delle più importanti e tutte le nuove istruzioni del 286 sono dedicate proprio alla scrittura di codice per la modalità protetta. Anche se l'argomento della modalità protetta è molto importante, in questo manuale non verrà approfondito poiché per quanto riguarda l'ottimizzazione del codice, la programmazione in modalità protetta non differisce dalla creazione di codice per la modalità reale. Le istruzioni per la modalità protetta vengono principalmente utilizzate dal sistema operativo.

Il concetto più importante nella programmazione in modalità protetta è il fatto che i registri di segmento non contengono indirizzi così come avviene in modalità reale. Come si ricorderà in modalità reale era possibile trovare l'indirizzo iniziale di un segmento moltiplicando per 16 (10 in esadecimale) il valore contenuto nel registro di segmento. Nella programmazione in modalità protetta il valore contenuto nel registro di segmento è chiamato selettore. Il selettore è un puntatore a una tabella che

contiene gli effettivi indirizzi iniziali dei segmenti. Questa tabella (ma ve ne possono essere molte) è chiamata tabella dei descrittori. La tabella dei descrittori contiene anche altre informazioni come ad esempio le dimensioni dei segmenti, il livello di protezione e il tipo di informazioni contenute nel segmento, ovvero codice o dati.

Quello che segue è un elenco delle nuove istruzioni introdotte dal 286 per la modalità protetta. In ogni caso non ne viene fornita una descrizione completa (per vedere alcuni esempi di programmi in modalità protetta si consulti il Capitolo 18).

arpl	Adjust Requested PriviLege
clts	CLear Task Switched flag
lar	Load Access Rights
lgdt	Load Global Descriptor Table
lidt	Load Interrupt Descriptor Table
lldt	Load Local Descriptor Table
lmsw	Load Machine Status Word
lsl	Load Segment Limit
ltr	Load Task Register
sgdt	Store Global Descriptor Table
sidt	Store Interrupt Descriptor Table
sldt	Store Local Descriptor Table
smsw	Store Machine Status Word
str	Store Task Register
verr	VERify Read
verw	VERify Write

7.3 L'80386

Il chip 80386 è dotato di numerosi miglioramenti. Innanzitutto è un microprocessore a 32 bit con supporto della memoria virtuale paginata. Inoltre il 386 introduce nuove istruzioni e nuove forme di istruzioni preesistenti. Infine è stato migliorato il tempo di esecuzione di molte istruzioni.

La memoria virtuale paginata è importante poiché consente di scrivere programmi che (almeno in teoria) possono essere eseguiti con qualsiasi quantità di memoria. Quando è necessario utilizzare più memoria, una porzione di memoria che recentemente non è stata utilizzata può essere trasferita su disco. Le pagine di memoria hanno una lunghezza di 4 KB. Tale funzionalità non è automatica ma deve essere esplicitamente richiamata dal sistema operativo. In particolare questa possibilità viene sfruttata sia da Windows che da OS/2.

Sul 386 i registri di utilizzo generale hanno tutti una lunghezza di 32 bit così come i bus per i dati e gli indirizzi. Il 386 SX ha un bus dati a 16 bit. Quando venne annunciato il 386 SX, per chiarezza il 386 venne ribattezzato 386 DX. Sia il 386 DX che il 386 SX possono utilizzare lo stesso software.

I registri del 386 sono tutti a 32 bit tranne i registri di segmento che rimangono a 16 bit. Questo consente di mantenere da un lato la compatibilità con i programmi per la modalità reale dei microprocessori 8088/8086 e nel contempo consente di utilizzare i

programmi per la modalità protetta. I dati contenuti nella tabella dei descrittori sono leggermente diversi rispetto al 286 e consentono di creare segmenti lunghi fino a 4 GB. Inoltre sono stati aggiunti due nuovi registri di segmento: FS e GS.

Ora i programmi possono utilizzare i registri a 16 bit (AX, BX e così via) oppure i nuovi registri a 32 bit (EAX, EBX e così via). I registri a 32 bit hanno conservato il nome dei registri a 16 bit ma preceduti dalla lettera "E". Il codice contenuto nei segmenti può utilizzare segmenti a 16 o 32 bit. Un prefisso dell'istruzione consente di utilizzare registri a 16 bit in un segmento a 32 bit e viceversa. Questo però può essere fonte di confusione ed è una considerazione molto importante quando si intende scrivere codice ottimizzato. Il funzionamento dei registri a 8 bit non viene invece influenzato. Ecco alcuni esempi:

```
mov    bx, cx          ; codici operativi esadecimali:  89 CB
```

Quando l'istruzione precedente viene assemblata e caricata in un segmento a 16 bit, i codici operativi esadecimali prodotti sono 89 CB. Se gli stessi codici operativi aparissero in un segmento a 32 bit, le istruzioni sarebbero:

```
mov    ebx, ecx        ; codici operativi esadecimali:  89 CB
```

Ma come può accadere questo? La tabella dei descrittori per il segmento di codice corrente (CS) carica, fra le altre cose, un bit che specifica se gli operandi sono a 16 o 32 bit. Le seguenti istruzioni si trovano in un segmento a 16 bit:

```
mov    b1, c1          ; codici operativi esadecimali:  88     CB
mov    bx, cx          ; codici operativi esadecimali:  89     CB
mov    ebx, ecx        ; codici operativi esadecimali:  66 89  CB
```

mentre le seguenti istruzioni si trovano in un segmento a 32 bit:

```
mov    b1, c1          ; codici operativi esadecimali:  88     CB
mov    bx, cx          ; codici operativi esadecimali:  66 89  CB
mov    ebx, ecx        ; codici operativi esadecimali:  89     CB
```

Il prefisso delle dimensioni degli operandi (66h) contenuto in un byte consente dunque di utilizzare registri o operandi in memoria diversi dalle dimensioni standard. L'assembler inserisce automaticamente tale prefisso senza alcun messaggio di avvertimento. L'assembler conosce infatti il tipo di segmento utilizzato sulla base dei parametri USE16 o USE32 nella direttiva SEGMENT. L'argomento verrà trattato da un esempio nei prossimi capitoli.

Nuove modalità di indirizzamento del 386

Precedentemente le modalità di indirizzamento erano costituite da qualsiasi combinazione di base, indice e scostamento. Il registro base poteva essere BX o BP e il registro indice poteva essere SI o DI. Nella modalità di indirizzamento a 32 bit del 386, gli indirizzi possono essere specificati con qualsiasi combinazione di un registro base, un registro indice (scalabile) e un valore di scostamento. La prima modifica è il fatto che il registro indice può essere moltiplicato per 1, 2, 4 o 8. La seconda modifica è il fatto che i registri base possono essere costituiti da qualsiasi registro di utilizzo

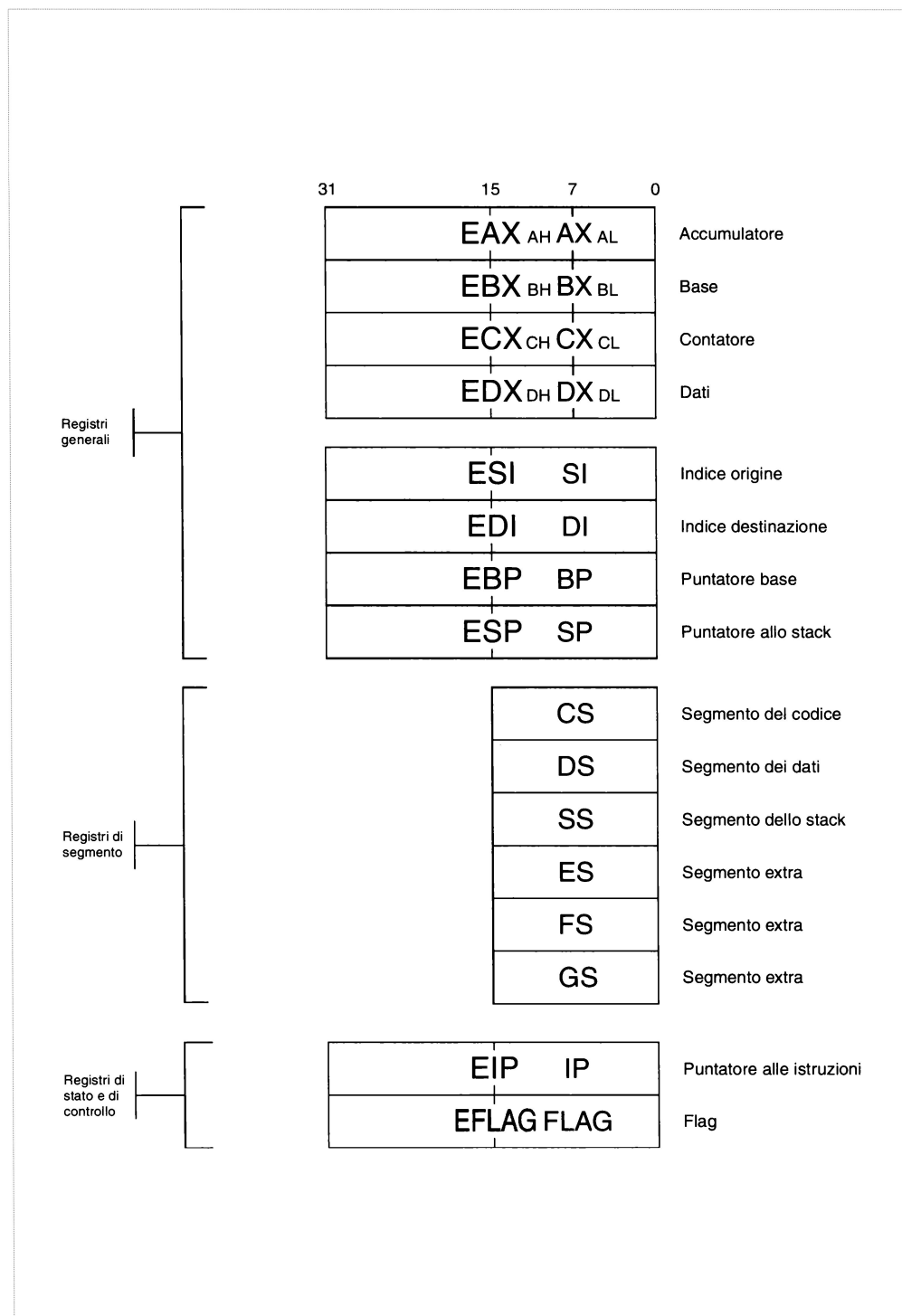



Figura 7.1 Schema dei registri a 32 bit.

generale (EAX, EBX, ECX, EDX, EDI, ESI e EBP). Naturalmente gli indirizzi sono tutti a 32 bit.

Nuove istruzioni del 386

Questa sezione descrive le istruzioni introdotte o modificate dal 386.

Scansione dei bit: BSF e BSR

Queste istruzioni eseguono la scansione di un operando alla ricerca del primo bit uguale a 1. BSF esegue la scansione in avanti mentre BSR esegue la scansione all'indietro. Quando viene trovato un bit uguale a 1, viene cancellato il flag zero (ZF) e viene restituita la destinazione specificando la posizione del primo bit trovato. Il valore 0 corrisponde al bit meno significativo. Ad esempio:

```
mov     eax, 84h
bsf     ebx, eax
jz      none
        ; ebx = 3
mov     eax, 84h
bsr     ebx, eax
jz      none
        ; ebx = 7
```

Test dei bit: BT, BTC, BTR e BTS

Le istruzioni di test dei bit copiano nel carry il valore contenuto in un determinato bit in modo da poterne eseguire il test con le istruzioni JC o JNC. BT (Bit Test) copia semplicemente il bit nel flag carry. BTC (Bit Test and Complement) inverte il bit specificato e quindi lo copia nel flag carry. BTR (Bit Test and Reset) copia il bit nel flag carry e quindi gli assegna il valore 0. BTS (Bit Test and Set) copia il bit nel flag carry e quindi gli assegna il valore 1. In tutte queste istruzioni, il primo operando (la destinazione) è un registro o un operando di memoria di cui deve essere eseguito il test. Il secondo operando (l'origine) deve essere un registro o un valore immediato corrispondente alla posizione del bit che deve essere copiato. Ad esempio:

```
mov     ax, 33h
...
bt      ax, 2    ; copia in CF il bit 2
jc      found
```

Conversione: CDQ e CWDE

CDQ (Convert Double to Quad) è la forma con operandi a 32 bit di CWD. CDQ converte la dword con segno contenuta in EAX in una quadword con segno contenuta in EDX:EAX, mantenendone il segno. CWDE (Convert Word to Double Extended) è la forma con operandi a 32 bit di CBW. CWDE converte una word con segno contenuta in AX in una dword contenuta in EAX, conservandone il segno.

Moltiplicazione con segno: IMUL

Vi sono due nuove forme dell'istruzione IMUL. La prima consente di moltiplicare due registri a 16 o 32 bit. La seconda consente di utilizzare come origine un operando in memoria. Poiché gli operandi di origine e di destinazione hanno le stesse dimensioni, è possibile che il risultato sia troppo esteso. In questo caso vengono impostati i flag carry e overflow. Ad esempio:

```
mov   ebx, 10
mov   ecx, 100000
imul  ecx, ebx
```

Salti condizionali

Sul 386 i salti condizionali possono essere near o short. Precedentemente era possibile eseguire solo salti short (da -128 a +127 byte). I salti near accettano valori compresi fra -32768 e +32767 byte.

LOOP

L'istruzione LOOP decrementa CX e salta all'indirizzo di destinazione nel caso in cui CX sia diverso da 0. Inoltre vi sono forme che consentono di eseguire il test del flag zero (ZF). Sul 386 in modalità a 32 bit viene utilizzato il registro ECX. Utilizzando il prefisso dimensionale dell'operando vi sono alcune nuove forme dell'istruzione LOOP:

Istruzione	Istruzione alternativa	Registro contatore 16 bit	Registro contatore 32 bit
loop	—	CX	ECX
loopw	—	CX	CX
loopd	—	ECX	ECX
loope	loopz	CX	ECX
loopew	loopzw	CX	CX
looped	loopzd	ECX	ECX
loopne	loopnz	CX	ECX
loopnew	loopnzw	CX	CX
loopned	loopnzd	ECX	ECX

MOV

Il 386 è dotato di nuovi registri speciali. Ora l'istruzione MOV accetta come operandi uno qualsiasi di questi registri di controllo, di debug e di test. Tali registri sono CR0, CR2, CR3, DR0, DR1, DR3, DR6, DR7, TR6 e TR7. Per informazioni sull'impiego di questi registri, si consulti la documentazione Intel.

Spostamenti: MOVSX e MOVZX

MOVSX (MOVe with Sign-eXtend) trasferisce un operando a 8 o 16 bit in un registro a 16 o 32 bit, copiando il bit del segno nella metà superiore della destinazione. MOVZX (MOVe with Zero eXtend) sposta un operando a 8 o 16 bit in un registro a 16 o 32 bit, inserendo una serie di 0 nella metà superiore della destinazione.

PUSH/POP

Sono state create nuove forme delle istruzioni PUSH e POP che consentono di utilizzare valori a 32 bit:

```
push    push di 2 o 4 byte sulla base della modalità operativa utilizzata (16 o 32 bit)
pushw   push di 2 byte
pushd   push di 4 byte
pushf   push dei flag a 16 bit
pushfd  push dei flag a 32 bit
pusha   push di tutti i registri a 16 bit
pushad  push di tutti i registri a 32 bit
pop     pop di 2 o 4 byte sulla base della modalità operativa utilizzata (16 o 32 bit)
popw    pop di 2 byte
popd    pop di 4 byte
popf    pop dei flag a 16 bit
popfd   pop dei flag a 32 bit
popa    pop di tutti i registri a 16 bit
popad   pop di tutti i registri a 32 bit
```

SET*cc* (impostazione condizionale)

Le istruzioni SET*cc* impostano l'operando specificato a 1 nel caso in cui la condizione verificata sia vera oppure a 0 se la condizione è falsa. I codici *cc* sono gli stessi dei salti condizionali, ad esempio Z per "uguale a 0" e NZ per "diverso da 0".

Scorrimenti doppi: SHLD e SHRD

Le istruzioni di scorrimento doppio consentono di far scorrere due operandi come se si trattasse di un'unica entità. Ad esempio:

```
shld   ax, dx, 1        ; scorrimento a sinistra di un numero a 16 bit in precisione doppia
```

corrisponde a:

```
shl    ax, 1            ; scorrimento a sinistra
rcl    dx, 1            ; rotazione con cary
```

L'eventuale riporto proveniente dal primo scorrimento (AX) viene trasferito nel bit inferiore di DX. Il bit alto di DX viene invece trasferito nel flag carry. Questo tipo di scorrimento è dunque simile a un ciclo. Ad esempio il doppio scorrimento a destra seguente:

```
shrd   [ebx3] eax, cl   ; scorrimento a destra di un numero a 32 bit in precisione doppia
```

equivale a:

```
  mov      ch, 0
lbl:
  shr      [ebx], 1
  rcr      eax, 1
  loopw    lbl
```

La modalità protetta

La modalità protetta funziona in linea di massima in questo modo: all'inizio la CPU si trova in modalità reale. Il sistema operativo o qualche altro programma di controllo attiva una tabella di descrittori formata da un elenco dei segmenti di codice e di dati con i relativi indirizzi ed altri attributi. Quindi la CPU viene portata in modalità protetta. Il controllo delle aree di memoria dedicate a un programma viene eseguito dal sistema operativo e dunque un programma non può erroneamente accedere alla memoria allocata da un altro programma. Quando un programma richiede più memoria, la richiesta passa sempre attraverso il sistema operativo. Il sistema operativo conserva informazioni sulla memoria utilizzata da ogni programma in esecuzione. Ogni comunicazione fra programmi viene controllata dal sistema operativo.

Poiché la modalità protetta è fondamentalmente dedicata alla realizzazione di sistemi operativi e poiché la maggior parte delle operazioni in modalità protetta viene gestita tramite chiamate al sistema operativo, l'argomento verrà introdotto nel Capitolo 18.

Parte terza

INTRODUZIONE AL PENTIUM E AI RELATIVI STRUMENTI DI SVILUPPO

Capitolo 8

L'80486 e il Pentium

Nel 1989 Intel annuncia il microprocessore 80486. Questo chip non ha portato significative modifiche al set di istruzioni.

8.1 Il 486

La popolarità dell'architettura Intel ha dunque determinato il futuro del set di istruzioni dei nuovi chip della famiglia 80x86 e le regole erano chiare:

1. non era necessario modificare le istruzioni principali;
2. anche se fossero state apportate modifiche, nessuno le avrebbe utilizzate, per garantirsi la compatibilità con la base installata;
3. l'aumento di prestazioni doveva venire principalmente da modifiche all'hardware e non al software.

Tuttavia, l'architettura interna del 486 è notevolmente diversa rispetto ai processori precedenti. Sono state impiegate tecniche di progettazione RISC, inclusa una pipeline a cinque fasi per le istruzioni e una certa quantità di memoria cache interna. Il coprocessore per i numeri in virgola mobile (che dall'8088 all'80336 era un chip distinto) è stato integrato nel microprocessore consentendo di ottenere un ulteriore aumento di prestazioni. Le sei istruzioni aggiunte vengono principalmente impiegate dal sistema operativo ma è opportuno discutere almeno le prime tre in dettaglio poiché possono essere utilizzate da qualsiasi programma applicativo. Le nuove istruzioni sono:

BSWAP	Byte SWAP
XADD	eXchange and ADD
CMPXCHG	CoMPare and eXCHanGe
INVD	INValidate Data cache
WBINVD	Write Back and INValidate Data cache
INVLPG	INValidate TLB (Translation Lookaside Buffer) entry

BSWAP

L'istruzione di scambio dei byte inverte l'ordine dei byte di un registro dword. L'istruzione può essere utilizzata per vari scopi ma è probabilmente stata aggiunta per convertire il formato dei dati per sistemi che memorizzano i valori partendo dal byte più significativo (come si ricorderà il chip della famiglia 80x86 memorizzano i dati partendo dal byte meno significativo). L'istruzione può essere utilizzata nel seguente modo:

```
bswap  eax
```

XADD

L'istruzione di scambio e somma è una combinazione delle istruzioni XCHG e ADD. A partire dal 486, queste due istruzioni possono essere riunite in un'unica istruzione che può essere preceduta dal prefisso LOCK. Presumibilmente l'istruzione ha lo scopo di aiutare il sistema operativo in ambienti multi-processore. L'operando di destinazione conterrà la somma dei due operandi mentre l'operando di origine conterrà il valore originale della destinazione. I flag vengono modificati così come avveniva nel caso dell'istruzione ADD. Gli operandi possono essere lunghi 8, 16 o 32 bit. L'operando di destinazione può essere un registro o un operando in memoria. L'operando di origine deve essere un registro. Ad esempio:

```
xadd   mem, eax
```

corrisponde a:

```
xchg   mem, eax
add    mem, eax
```

oppure a:

```
mov    tmp_reg, eax
mov    eax, mem
add    tmp_reg, eax
mov    mem, tmp_reg
```

Vi sono tre motivi per preferire XADD. Il primo è legato allo scopo per il quale è stata creata l'istruzione, ovvero l'utilizzo in ambienti multi-processore o la comunicazione fra più processori. Il secondo è legato a considerazioni sulla compattezza del codice. Infine, sul 486 questa istruzione è più veloce delle due istruzioni equivalenti ma occorre notare che sul Pentium XADD è più lenta rispetto alle due istruzioni che la compongono.

CMPXCHG

L'istruzione di confronto e scambio è simile a XADD e anch'essa è dedicata all'impiego in sistemi operativi multi-processore. Ancora una volta, per assicurare l'integrità del sistema durante l'esecuzione dell'istruzione può essere utilizzato il prefisso LOCK che blocca il bus del processore. Gli operandi sono gli stessi di XADD. Tuttavia, nel caso di CMPXCHG, l'accumulatore (AL, AX o EAX) è sempre un operando implicito. Il suo funzionamento è piuttosto particolare e dunque ecco un esempio:

```
cmpxchg  mem, ebx
```

Ecco cosa accade in questo esempio:

- EAX viene confrontato con mem (come in CMP eax, mem);
- se sono uguali, EBX viene copiato in mem (come in MOV mem, ebx);
- se sono diversi, mem viene copiato in EAX (come in MOV eax, mem).

Ecco dunque il codice equivalente per la forma generale CMPXCHG dest, src:

```
    cmp  eax, dest
    jne  not_equal
    mov  dest, src
    jmp  done
not_equal:
mov     eax, dest
done:
```

8.2 **Il Pentium**

Nel 1993 Intel annuncia il processore Pentium. Sono numerose le modifiche che rendono il Pentium molto più veloce rispetto al 486. La modifica principale riguarda la presenza delle due pipeline che consentono al Pentium di eseguire simultaneamente due istruzioni. Inoltre sono state migliorate le prestazioni di molte istruzioni, in particolare delle istruzioni in virgola mobile. Infine è stata aggiunta una funzione di previsione della destinazione dei salti per eliminare i ritardi legati a questa operazione. Le altre modifiche apportate hanno invece lo scopo di ridurre gli inevitabili colli di bottiglia che si verificano quando si eseguono più istruzioni ad una frequenza più elevata. Ecco un breve elenco delle modifiche apportate al chip.

1. Bus a 64 bit.
2. Cache per il codice di 8 KB e cache per i dati di altri 8 KB (sul 486 era presente una sola cache comune da 8 KB).
3. Un minor numero di cicli di clock per alcune istruzioni (specialmente quelle in virgola mobile).
4. Sistema di previsione dei salti.
5. Doppia pipeline.
6. Frequenze di clock più elevate.

In sostanza il Pentium è dotato di un'architettura superscalare con pipeline. Con superscalare si intende la possibilità della CPU di eseguire due o più istruzioni per ciclo (per la precisione il Pentium può generare il risultato di due istruzioni in un unico ciclo di clock). Con architettura pipeline si fa riferimento al fatto che la CPU è in grado di eseguire ogni porzione di un'istruzione in fasi diverse. Quando viene completata la prima fase, può essere caricata l'istruzione successiva mentre quella corren-

te passa alla seconda fase. L'80486 e il Pentium hanno pipeline a cinque fasi. Il Pentium ha poi due pipeline, chiamate pipe U e pipe V.

Ovviamente il problema nel caso di più pipeline è il fatto che alcune istruzioni possono bloccare l'avanzamento di altre istruzioni nella pipeline a causa di conflitti nell'uso dei registri o nella generazione degli indirizzi. Questo problema verrà discusso in dettaglio nel Capitolo 10.

La memoria cache del Pentium

La memoria cache del Pentium è costituita da 8 KB per il codice e 8 KB per i dati mentre nel 486 vi erano solamente 8 KB per entrambi gli usi (alcune nuove versioni del 486 sono dotate di una cache combinata di 16 KB). La possibilità di utilizzare cache distinte per il codice e i dati offre numerosi vantaggi. Infatti la lettura delle istruzioni avviene nella cache del codice mentre la lettura e scrittura dei dati avviene in una cache distinta. Questa separazione genera un minor numero di conflitti nel bus interno e dunque elimina i ritardi introdotti da tali conflitti. Ma l'importanza della doppia cache non si ferma qui: questo sistema consente di conservare nella cache del codice utili informazioni relative a ogni byte del codice stesso. Infatti il Pentium conserva varie informazioni sulle possibilità di accoppiamento di ogni singola istruzione. Tale argomento verrà discusso nei prossimi capitoli.

Nuove istruzioni del Pentium

Il Pentium introduce sei nuove istruzioni e alcune nuove forme dell'istruzione MOV:

CMPXCHG8B - CoMPare and eXCHange 8 Byte

L'istruzione CMPXCHG8B è molto particolare: è simile all'istruzione CMPXCHG introdotta a partire dal 486 ma confronta 8 byte specificando un solo operando. EDX:EAX e ECX:EBX sono infatti operandi impliciti. EDX:EAX viene confrontato con l'operando in memoria. Se sono uguali, il valore ECX:EBX viene salvato in memoria mentre in caso contrario il valore in memoria viene copiato in EDX:EAX.

CPUID - CPU IDentification

L'istruzione CPUID restituisce informazioni relative alla CPU in modo che un programma possa determinare le funzionalità offerte dal chip. Per poter utilizzare questa istruzione è però necessario verificare se il sistema è dotato di un microprocessore meno recente. Fortunatamente Intel consente di rilevare se è disponibile questa istruzione. Infatti vi è un nuovo bit nel registro EFLAGS, ovvero il bit 21. Se un programma può modificare lo stato di questo bit, significa che il microprocessore supporta l'istruzione CPUID. Ad esempio la nuova istruzione è supportata dalle nuove versioni di 386 e 486. Ecco come funziona l'istruzione:

```
mov    eax, 0
cpuid
```

All'uscita, EAX contiene il massimo valore utilizzabile in EAX. Il massimo valore attualmente utilizzabile è 1. I registri EBX, ECX ed EDX restituiscono la stringa di identificazione del produttore ovvero "GenuineIntel":

EBX	= 'Genu'	('G' nel registro BL)
EDX	= 'inel'	('i' nel registro DL)
ECX	= 'ntel'	('n' nel registro CL)

Quando in EAX viene utilizzato il valore 1, vengono restituite le seguenti informazioni:

EAX	bit 0-3	Codice identificativo (0Bh o superiore)
EAX	bit 4-7	Numero modello (1 o superiore)
EAX	bit 8-11	Numero della famiglia (5 per il Pentium)
EAX	bit 12-31	riservati
EBX		reservato (0)
ECX		reservato (0)

I flag di EDX hanno il seguente significato:

EDX	bit 0	1 = FPU sul chip
EDX	bit 1-6	non documentati
EDX	bit 7	1 = eccezione di verifica della macchina
EDX	bit 8	1 = istruzione CMPXCHG8B
EDX	bit 9-31	riservati

RDTSC ReaD Time Stamp Counter

Si tratta di un'istruzione curiosa poiché è potenzialmente molto utile ma non è documentata da Intel. Intel l'ha indicata nella mappa dei codici operativi e nell'elenco delle nuove istruzioni del Pentium Processor User's Manual ma non specifica come utilizzarla. Ecco dunque come funziona. Ad ogni ciclo di CPU viene incrementato un contatore a 64 bit. RDTSC restituisce questo valore in EDX:EAX. Il codice operativo di RDTSC è 0F 31.

RDMSR - ReaD from Model Specific Register

L'istruzione RDMSR restituisce nei registri EDX ed EAX informazioni specifiche relative al Pentium. Nel registro ECX viene caricato un valore che specifica le informazioni che devono essere restituite. Intel documenta i due parametri seguenti:

ECX	Nome registro	Descrizione
0	Machine check address	Indirizzo del ciclo che ha provocato l'eccezione
1	Machine check type	Tipo del ciclo che ha provocato l'eccezione

WRMSR - WRite to Model Specific Register

L'istruzione WRMSR è l'inversa di RDMSR. I valori documentati da Intel per WRMSR sono gli stessi di RDMSR.

RSM - Resume from System Management mode
L'istruzione RSM esce dalla modalità System Management (SMM). Questa modalità consente al software di gestire l'alimentazione del sistema e/o le funzioni di sicurezza in modo trasparente ai programmi applicativi e ai sistemi operativi. L'ingresso nella modalità SMM avviene tramite un segnale hardware; l'istruzione fa in modo che venga eseguito il codice contenuto in un altro spazio di indirizzamento. La modalità SMM è come la modalità reale ma con uno spazio di indirizzamento di 4 GB.

8.3 Riepilogo

Questo è tutto. Il presente capitolo conclude la descrizione dei set delle istruzioni. Anche se in questo capitolo sono state introdotte molte novità, vi sono due concetti particolarmente importanti: la pipe U e la pipe V.

A partire dal prossimo capitolo verranno descritti i modi in cui è possibile eseguire contemporaneamente due istruzioni, una per ogni pipeline del Pentium.

Capitolo 9

Programmazione superscalare

Questo capitolo e i successivi descrivono le caratteristiche che rendono il Pentium un microprocessore così potente. Fino ad ora i concetti presentati potevano risultare, almeno vagamente, familiari. Dunque, chi non avesse esperienza diretta nella realizzazione e debugging di programmi assembler, può tornare a rileggere i capitoli precedenti per chiarire meglio i concetti esposti.

Vi sono tre funzionalità specifiche che rendono la programmazione del Pentium così diversa rispetto al 386 e al 486:

- l'architettura superscalare con pipeline;
- la previsione della destinazione dei salti;
- l'ottimizzazione dei tempi di esecuzione dei cicli.

In questo capitolo verranno descritti questi tre argomenti, partendo dall'architettura superscalare

9.1 Le due pipeline intere

Il processore Pentium è dotato di due pipeline intere: la pipe U e la pipe V. La pipe U è in grado di eseguire qualsiasi istruzione intera. La pipe V può invece eseguire solo istruzioni semplici. Quando nella coda di prefetch si trovano in successione due istruzioni semplici e si verificano determinate condizioni, la CPU "accoppia" tali istruzioni e inizia contemporaneamente la loro esecuzione. Quando un processore è dotato di due o più pipeline in parallelo si dice che ha un'architettura superscalare.

Uno dei punti chiave nell'ottimizzazione del Pentium consiste nel conoscere e seguire il più possibile le regole di accoppiamento delle istruzioni.

La possibilità di accoppiare due istruzioni è determinata dal Pentium nella seconda fase della pipeline. Infatti vi sono due decoder paralleli che tentano di decodificare ed eseguire le due istruzioni successive. Questa determinazione si basa su una serie di regole, descritte nella Figura 9.1.

Istruzioni semplici

Precedentemente si è detto che la pipe V in grado di eseguire solo istruzioni semplici. Le istruzioni semplici costituiscono un determinato sottoinsieme del set di istruzioni della famiglia 80x86. Fondamentalmente le istruzioni semplici sono MOV, le operazioni aritmetiche e logiche (ad esempio ADD, SUB, CMP, AND, OR e così via), INC, DEC, PUSH, POP, LEA, NOP, gli scorrimenti, CALL, JMP e i salti condizionali. La Tabella 9.1 contiene un elenco completo di tutte le istruzioni semplici e i relativi formati.

Vi sono anche alcune istruzioni che non appaiono in questo elenco anche se vengono comunemente considerate istruzioni "semplici"; si tratta delle istruzioni che alterano il registro dei flag (STC, CLC, CMC e così via), delle istruzioni XCHG, delle conversioni di tipo (CBW) e infine di NOT e NEG. Probabilmente Intel ha scelto di inserire in questo elenco le istruzioni più frequentemente utilizzate, tenendo in considerazione la difficoltà di implementazione (il numero di transistor necessari nel chip).

A parte la necessità che entrambe le istruzioni siano semplici, vi sono anche altri requisiti. Alcuni hanno a che fare con la necessità di evitare conflitti fra le due istruzioni accoppiate. Ad esempio, due istruzioni possono essere accoppiate se entrambe leggono lo stesso registro ma non se entrambe scrivono lo stesso registro. Altri requisiti di accoppiamento hanno a che fare con alcune limitazioni progettuali del Pentium. La Figura 9.1 presenta un elenco completo di queste limitazioni.

Tabella 9.1 Istruzioni semplici.

MOV	reg, reg	
MOV	reg, mem	
MOV	reg, imm	
MOV	mem, reg	
MOV	mem, imm	
alu	reg, reg	
alu	reg, mem	
alu	reg, imm	
alu	mem, reg	
alu	mem, imm	dove alu=add, adc, and, or, xor, sub, sbb, cmp e test
INC	reg	
INC	mem	
DEC	reg	
DEC	mem	
PUSH	reg	
POP	reg	
LEA	reg, mem	
JNP	near	
CALL	near	
Jcc	near	
NOP		
scorrimento	reg	dove scorrimento = sal, sar, shl, shr, rcl, rcr, rol, ror
scorrimento	mem,1	
scorrimento	reg, imm	
scorrimento	mem, imm	

Note:
- rcl e rcr non sono accoppiabili se la rotazione si estende per più di una posizione.
- Le istruzioni con operandi memoria, immediato (mem, imm) non possono essere accoppiate quando esiste uno scostamento nell'operando mem.
- Le istruzioni con registri di segmento non sono accoppiabili.

1. Entrambe le istruzioni devono essere semplici.
2. Gli scorrimenti e le rotazioni possono essere eseguiti solo nella pipe U.
3. Le istruzioni ADC e SBB possono essere eseguite solo nella pipe U.
4. Le istruzioni JMP/CALL/Jcc possono essere eseguite solo nella pipe V.
5. Nessuna delle due istruzioni può contenere uno scostamento e un operando immediato (una costante).
6. Le istruzioni con prefisso possono essere eseguite solo nella pipe U (tranne il prefisso 0F di Jcc).
7. Le istruzioni della pipe U devono occupare un solo byte o non possono essere accoppiate se non a partire dalla seconda volta (nella quale vegono eseguite dalla memoria cache).
8. Fra le istruzioni non vi possono essere dipendenze di tipo "scrittura-lettura" o "scrittura-scrittura" sui registri tranne nel caso speciale del registro dei flag e del puntatore allo stack (regole 9 e 10).
9. L'eccezione del registro dei flag consente di accoppiare un'istruzione CMP o TEST con un'istruzione Jcc anche se CMP/TEST scrive i flag e Jcc legge i flag.
10. L'eccezione del puntatore allo stack consente di accoppiare due PUSH o due POP anche se entrambe leggono e scrivono il registro SP (o ESP).

Figura 9.1 Regole di accoppiamento delle istruzioni.

Già utilizzando le regole da 1 a 4 e la regola 8 è possibile ottenere ottimi risultati. Ma vi sono moltissime regole da tenere in considerazione nella creazione del codice. Questo è il motivo per il quale è stato sviluppato il programma PENTOPT descritto nel Capitolo 11 e incluso nel disco fornito con questo manuale.

Note sulle regole di accoppiamento

In questa sezione verranno descritte più in dettaglio le regole di accoppiamento. Le prime quattro regole sono autoesplicative e dunque si parlerà solo delle regole dalla quinta in poi.

La regola 5 si basa sul numero dei componenti dell'istruzione e/o degli operandi che i decoder del Pentium possono elaborare per determinare se due istruzioni possono essere accoppiate. Si tratta di una regola con la quale occorre sempre fare i conti. Gli operandi in memoria possono essere costituiti da vari componenti (un registro base, un registro indice scalato e uno scostamento); lo scostamento è un valore costante che definisce l'offset e che deve essere aggiunto agli altri due registri. Ogni componente è opzionale ma vi deve essere almeno uno dei tre componenti. Un operando immediato è una costante che si trova nella porzione di origine di un'istruzione.

Ad esempio:

```
mov   [bx], 1         ; solo base             OK
mov   [ebx+esi], 2    ; base + indice         OK
mov   [ebx+2], 2      ; base + scostamento    non accoppiabile
mov   var1, 4         ; solo scostamento      non accoppiabile
```

La regola del byte di prefisso (regola 6) è importante principalmente in due casi. Il primo si verifica quando si utilizza l'uscita dal segmento. Gli assembler MASM e TASM inseriscono automaticamente le istruzioni di uscita dal segmento sulla base dei parametri della direttiva ASSUME. Il secondo caso si verifica quando si scrive codice misto a 16 e 32 bit. I prefissi REP, REPE e REPNE non possono essere utilizzati su tutte le istruzioni semplici. Il prefisso LOCK può essere utilizzato solo con alcune istruzioni aritmetico-logiche.

A causa della regola del byte singolo (regola 7) le uniche istruzioni che possono essere accoppiate fin dalla prima esecuzione sono INC/DEC reg, PUSH/POP reg e NOP. Questo non presenta grossi problemi nelle applicazioni poiché raramente si esegue l'ottimizzazione di codice eseguito una sola volta. È molto più importante ottimizzare il codice eseguito ripetutamente. Ma questo significa anche che le istruzioni possono risultare accoppiate in modo diverso nella prima e nelle successive esecuzioni. Ad esempio:

```
              Prima  Successive
mov   ax, 1   1      1           (cicli di CPU)
inc   bx      2      1
mov   cx, 1   2      2
call  xyz     3      2
```

La logica che determina le dipendenze lettura/scrittura (regole 8, 9 e 10) si basa sul fatto che i registri sono entità a 32 bit. Pertanto una lettura o scrittura su una parte di un registro equivale a utilizzare l'intero registro. Dunque anche scrivendo su AL, AH o AX si scriverà comunque su EAX. E, anche se Intel è un po' vaga nella descrizione dell'accoppiamento di istruzioni che modificano i flag, l'esperienza personale insegna che tutte le istruzioni semplici alu/INC/DEC possono essere accoppiate a dei salti condizionali. Questo presenta interessanti possibilità di ottimizzazione per il fatto che, se possibile, si dovrebbero sempre utilizzare le istruzioni CMP o TEST per impostare i flag in quanto tali istruzioni modificano solo il registro dei flag. Ad esempio, per verificare se AX=0 si possono utilizzare le tre istruzioni seguenti:

```
cmp   ax, 0
or    ax, ax
test  ax, ax
```

L'istruzione CMP è lunga 3 byte e le altre 2 byte. L'istruzione OR scrive su AX riducendo le possibilità di accoppiamento.

Ecco altri esempi di dipendenze di tipo "scrittura/lettura":

```
scrittura/lettura (non accoppiabili)    mov   al, 1
                                        add   bh, ah
                                        mov   ax, 1
                                        add   bx, ax
```

scrittura/scrittura (non accoppiabili)	mov	eax, 1
	add	eax, ebx
	mov	ax, 1
	mov	ax, 2
lettura/scrittura (accoppiabili)	mov	ax, bx
	inc	bx
lettura (accoppiabili)	mov	eax, ebx
	add	ecx, ebx

9.2 La logica di previsione dei salti

La previsione dei salti è una funzionalità introdotta dal Pentium. Quando si incontra un'istruzione di salto o una chiamata, l'indirizzo dell'istruzione viene utilizzato per accedere al buffer BTB (Branch Target Buffer) che cerca di prevedere il risultato dell'istruzione. Non è possibile interveire molto sulla logica di previsione dei salti poiché è completamente automatica. Molte istruzioni JMP o CALL possono essere eseguite in un unico ciclo se la logica di previsione ottiene sempre risultati corretti.

Ecco, sommariamente, il funzionamento della logica di previsione dei salti. Il Pentium conserva nel buffer BTB il risultato degli ultimi 256 salti e tenta di prevedere la destinazione di ogni JMP o CALL. A tale scopo conserva una registrazione dei risultati precedenti di un salto per determinare se è stato eseguito o meno. Se la previsione è corretta, un salto condizionale richiederà un solo ciclo di CPU.

Nel Pentium vi sono due code di prefetch, entrambe lunghe 32 byte. La logica di previsione dei salti si verifica nella fase D1 (la seconda fase) della pipeline nella quale si prevede se verrà eseguito o meno un salto e la sua destinazione. Quando viene previsto un salto, l'altra coda di prefetch inizia a leggere istruzioni. Se la previsione si rivela errata, le code vengono vuotate e viene riattivato il prefetch. Fino alla CPU 486, la condizione migliore nel caso dei salti condizionali era quella di non eseguire mai salti. Nel caso del Pentium la migliore ottimizzazione si ottiene mantenendo un comportamento costante, ovvero eseguendo sempre i salti oppure non eseguendoli mai. Sui processori fino al 486 dunque il codice era più veloce quando non veniva eseguito alcun salto. In generale questo è vero anche sul Pentium ma, quando in un ciclo si determina che viene sempre eseguito un salto, la velocità ottimale si ottiene proprio con altri salti mentre un'esecuzione lineare provocherà un certo ritardo. Tale ritardo non è trascurabile come si può vedere nella Tabella 9.2

Tabella 9.2 Ritardi determinati da errate previsioni sui salti.

Istruzione	Pipe U	Pipe V
Salti condizionali	4	5
JMP	3	3
CALL	3	3

Quando si supera il limite di 256 salti? Innanzitutto, durante un interrupt hardware che si verifica in un ciclo molto stretto è possibile superare il limite di 256 salti prima che venga restituito il controllo. Inoltre, durante una commutazione di task in un ambiente multitasking, ogni commutazione imporrà il riavvio del task e dunque la generazione di un nuovo buffer per i salti. Non vi è nulla che si può fare nei programmi applicativi per ridurre questo genere di ritardi.

Ecco un esempio in cui è possibile provocare inavvertitamente un overflow della tabella dei salti:

```
                ; con e senza previsione dei salti
                ; (i cicli fanno riferimento al caso di un carattere diverso dallo spazio)
loop1:
  mov  al, [si] ;  1   1
  inc  si       ;  0   0 (0 grazie all'accoppiamento delle istruzioni)
  cmp  al, ' '  ;  1   1
  jne  foo      ;  0   3
  call space
  imp  bar
foo:
  inc  dx       ;  1   1
bar:
  dec  cx       ;  0   0
  jnz  loop1    ;  1   3
       ; totale    4  10
```

Il codice precedente esegue la scansione di una stringa di lunghezza nota. Quando viene trovato uno spazio, viene richiamata una funzione, altrimenti viene incrementato un contatore. Quando viene trovato uno spazio, se la funzione di gestione dello spazio è piccola, la successiva iterazione del codice di loop1 verrà eseguita in 4 cicli, altrimenti richiederà 10 cicli. Si tratta di una differenza notevole ma in alcuni casi si ottengono risultati anche peggiori. Tuttavia, per modificare completamente il buffer BTB la funzione di gestione dello spazio deve essere composta da almeno varie centinaia di istruzioni. I 6 cicli aggiuntivi per la successiva iterazione di loop1 sono dunque insignificanti.

Ma ecco perché è così importante comprendere il funzionamento del sistema. Si supponga di voler controllare il tempo di esecuzione del codice contenuto nel ciclo loop1 ma non della funzione di gestione dello spazio e si supponga di utilizzare un dispositivo di misurazione hardware o un qualunque altro metodo. Si immagini di modificare la funzione di gestione dello spazio eliminando vari salti. Il risultato è che il codice di loop1 sembrerà essere più veloce.

9.3 **Ottimizzazione dei cicli**

Come ogni membro della famiglia 80x86 (dopo l'8088/8086), il Pentium presenta istruzioni che operano con un numero di cicli inferiore rispetto ai predecessori. Naturalmente ciò significa che i programmi verranno automaticamente eseguiti più velocemente rispetto ai precedenti membri della famiglia 80x86. Ma non sempre que-

sto è vero. In particolare è necessario sapere quali sono le istruzioni ottimizzate e il modo in cui devono essere posizionate all'interno del programma. Le combinazioni di istruzioni più veloci nel passato non sempre si rivelano altrettanto veloci sul Pentium. Si possono trovare esempi specifici negli ultimi capitoli e nell'Appendice B.

Il miglioramento più significativo sul Pentium (in termini di cicli di CPU) è costituito dalle prestazioni in virgola mobile. Le moltiplicazione e le somme vengono tutte completate in un massimo di 3 cicli. Quando la precisione lo consente, è preferibile eseguire una moltiplicazione per l'inverso di un numero invece di eseguire una divisione, ovvero è preferibile moltiplicare per 0.1 anziché dividere per 10. Su un 486 una moltiplicazione richiede 11-16 cicli e una divisione 73-89 cicli. Dunque una moltiplicazione può essere 6 o 7 volte più veloce. Sul Pentium una moltiplicazione richiede da 1 a 3 cicli e una divisione 39 cicli. Dunque la moltiplicazione è da 13 a 39 volte più veloce. Ovviamente questa considerazione può aumentare notevolmente la velocità delle operazioni in virgola mobile.

Un ulteriore aumento di prestazioni è legato al fatto che le l'istruzione FXCH può essere accoppiata con le più comuni istruzioni in virgola mobile. Per un esempio si consulti il Capitolo 16.

La Tabella 9.3 contiene un elenco delle più significative variazioni nei tempi di esecuzione, in termini di cicli, del Pentium rispetto al 386 e al 486.

Tabella 9.3 Cicli di esecuzione: le variazioni più significative.

		386	486	Pentium
add	reg, reg	2	1	1 (anche altre operazioni alu)
add	mem, reg	2	1	1
inc	reg	2	1	1 (anche dec)
inc	mem	2	1	1
mov	reg, reg	2	1	1
mov	mem, reg	2	1	1
mul		9-41	13-2	10-11
nop		3	3	1
pop	reg	4	1	1
push	reg	2	1	1
popa		24	9	5
pusha		18	11	5
ret		11	5	2
jcc		3/7	1/3	1*
imp near		8/9	3/5	1*
call near		8	3	1*
	(* = in caso di previsione corretta dei salti)			
loop		13	6/9	7/8
lods		5	5	2
rep movs		4	3	1
rep stos 5		4	1	
repe/ne cmps		9	7	4
repe/ne scas		8	5	4
fadd		23-72	8-32	1-3
fmul		29-82	11-16	1-3
fcos, fsin		123-772	257-354	1-126
fdiv		88-128	73-89	39

Capitolo 10

Funzionamento delle pipeline intere e in virgola mobile

Questo capitolo descrive l'utilizzo delle pipeline del Pentium da parte delle istruzioni intere e in virgola mobile. In particolare è importante comprendere il funzionamento delle pipeline per capire perché alcune istruzioni o combinazioni di istruzioni sono più efficienti rispetto ad altre. Innanzitutto si può partire dai motivi che spingono a creare una pipeline, per poi passare a descrivere la semplice pipeline utilizzata dal 486. Quindi verrà descritto il funzionamento delle pipeline del Pentium.

10.1 Lettura delle istruzioni (fetch)

Ogni istruzione deve essere letta (fetch) dalla memoria. Questo processo può essere uno dei colli di bottiglia più importanti nel sistema. Infatti è possibile che la CPU sia in grado di eseguire le istruzioni più velocemente rispetto alla velocità con la quale legge le istruzioni dalla memoria. Per eliminare questo collo di bottiglia, i processori della famiglia 80x86 sono sempre stati dotati di una coda di processi. Con fetch si intende l'operazione automatica di lettura del byte o dei byte che compongono la

prossima istruzione che deve essere eseguita. La coda di prefetch è un piccolo buffer FIFO (First In First Out) contenuto nella CPU. A tale proposito si consulti la Tabella 10.1.

10.2 La memoria cache

Quando fu progettato il 486, la CPU era così veloce che per poter leggere le istruzioni ad una velocità sufficiente venne introdotta una memoria cache interna. La memoria cache è un piccolo blocco di memoria ad alta velocità che contiene le parti più attive della memoria di sistema. La logica di funzionamento dei circuiti cache è fatta in modo tale che quando la CPU legge un byte da un'area di memoria, nella cache viene copiato un intero blocco di 32 o 64 byte. Tale operazione viene eseguita sulla base del fatto che la CPU probabilmente richiederà anche i byte successivi. Questo almeno finché la CPU non esegue un'istruzione di chiamata o un salto. Ma la cache è progettata in modo tale da gestire questa situazione poiché è in grado di conservare vari blocchi indipendenti di memoria. L'uso di ogni blocco di cache viene seguito sulla base dell'algoritmo più recentemente utilizzato. Quando la cache è completamente piena, viene scartato e dunque riutilizzato il blocco meno recentemente utilizzato.

Il 486 è dotato di 8 KB di memoria cache interna che contiene sia codice che dati. I nuovi chip 486DX4 sono però dotati di una cache da 16 KB. Anche il Pentium ha una memoria cache da 16 KB ma suddivisi in due blocchi distinti: 8 KB per il codice e 8 KB per i dati.

10.3 Le pipeline

Anche l'istruzione più semplice richiede che la CPU esegua vari passi indipendenti. Ecco cosa avviene nella CPU durante l'esecuzione di alcune istruzioni.

Tabella 10.1 Dimensioni della coda di prefetch.

Processore	Dimensioni della coda di prefetch	Note
8088	4	
8086	6	
80188	4	
80186	6	
286	8	
386	16	Molti sistemi hanno una cache esterna.
486	32	8 KB (o 16 KB) di cache interna per il codice e i dati.
Pentium	2 x 32	Cache di 8 KB per il codice e 8 KB per i dati.

Per caricare i dati dalla memoria, ad esempio con l'istruzione:

```
mov    ax, [bx]
```

la CPU deve eseguire le seguenti azioni:

- leggere (fetch) l'istruzione;
- decodificare l'azione da svolgere;
- calcolare l'indirizzo effettivo utilizzando la formula (DS * 16) + BX;
- leggere i dati dalla memoria;
- salvare i dati nel registro AX.

Ecco un'altra semplice istruzione.

```
add    ax, bx
```

Per questa istruzione, il processore deve:

- leggere (fetch) l'istruzione;
- decodificare l'azione da svolgere;
- sommare i due registri;
- salvare i dati nel registro AX.

Ed infine ecco un'ultima istruzione:

```
in byte ptr [bx+2]
```

Per questa istruzione il processore deve:

- leggere (fetch) l'istruzione;
- decodificare l'azione da svolgere;
- calcolare l'indirizzo effettivo utilizzando la formula (DS * 16) + BX + 2;
- leggere il byte dalla memoria;
- sommare 1;
- salvare i dati in memoria.

Non tutte le istruzioni eseguono gli stessi passi anche se tali passi sono in genere molto simili. Ma un computer potrebbe essere realizzato sul modello delle catene di montaggio: ogni istruzione si sposterebbe da un punto al successivo come se si trovasse su un nastro trasportatore. In ogni punto di lavoro vi sarebbe poi uno specialista che esegue il proprio lavoro. Questo è esattamente il funzionamento di una pipeline. Questo sistema è stato impiegato da Intel a partire dal 486. Le stazioni di lavoro o, per meglio dire, le fasi sono:

PF lettura (prefetch);
D1 decodifica dell'istruzione;
D2 generazione dell'indirizzo;
EX esecuzione e accesso alla memoria cache;
WB scrittura.

Ecco cosa avviene durante ogni fase della pipeline:

PF: Le istruzioni vengono lette dalla memoria cache o dalla memoria convenzionale e vengono salvate nella coda di prefetch.

D1: Le istruzioni vengono decodificate e suddivise in componenti, ovvero il codice operativo e gli operandi. Per le istruzioni che contengono un prefisso è richiesto un ciclo aggiuntivo.

D2: Viene calcolato, se presente, l'indirizzo effettivo dell'operando in memoria. Sul 486 è richiesto un ciclo aggiuntivo quando un indirizzo contiene sia una base che un indice oppure sia uno scostamento che un valore immediato.

EX: Il processore esegue le azioni richieste dall'istruzione, inclusa la lettura dei dati dalla memoria e il salvataggio dei risultati nei registri.

WB: L'istruzione è completata e i dati da scrivere in memoria vengono inviati al buffer di scrittura. Le istruzioni possono modificare lo stato della CPU.

Normalmente le istruzioni procedono da una fase alla successiva in un ciclo e dunque, sul 486 anche l'istruzione più veloce richiede cinque cicli. Ma chi ha avuto modo di osservare i cicli di esecuzione delle istruzioni ha potuto notare che molte istruzioni, sul 486, richiedono un solo ciclo. Come può essere?

I tempi indicati sono in effetti il minor numero di cicli richiesti da un'istruzione quando si trova insieme ad altre istruzioni. Quindi questo corrisponde all'effettiva velocità di elaborazione dell'istruzione e non al tempo di permanenza dell'istruzione nella pipeline. In una catena di montaggio può uscire una vettura ogni due minuti ma la realizzazione di ogni singola vettura può richiedere molte ore. Ecco un esempio (vedere la Tabella 10.2).

La maggior parte delle volte le istruzioni completano ogni fase della particolari in un ciclo, ad esclusione, a volte, della fase di esecuzione. Un'istruzione "complessa" può richiedere anche due o più cicli di esecuzione. In questo caso tutte le istruzioni che si trovano alle fasi precedenti verranno sospese (si ferma il "nastro trasportatore") fino al termine della fase di esecuzione dell'istruzione corrente. Ma fortunatamente la CPU è in grado di avviare le istruzioni che si trovano alle fasi precedenti (ovvero vi sono due "nastri trasportatori", uno che si ferma prima della fase di esecuzione e un secondo comprendente le ultime due fasi). La Tabella 10.3 mostra un caso in cui un'istruzione entra in stallo, rimanendo in fase di esecuzione per quattro cicli.

Tabella 10.2 Normale funzionamento della pipeline.

			Fasi			
Cicli	**PF**	**D1**	**D2**	**EX**	**WB**	
1	I1					Inizio I1
2	I2	I1				Inizio I2
3	I3	I2	I1			Inizio I3
4	I4	I3	I2	I1		
5	I5	I4	I3	I2	I1	Fine I1
6	I6	I5	I4	I3	I2	Fine I2
7		I6	I5	I4	I3	Fine I3

```
                                ; cicli
        lea     bx, tbl1        ; I1    1
lbl:    mov     al, [si]        ; I2    1
        xlat                    ; I3    4
        inc     si              ; I4    1
        cmp     al, dl          ; I5    1
        je      lbl             ; I6    1
```

Dunque quando un'istruzione non completa una fase in un ciclo, si dice che la pipeline entra in stallo. Nel listato seguente sono indicate alcune operazioni che possono provocare lo stallo illustrato nella Tabella 10.3.

10.4 I blocchi AGI (Address Generation Interlock)

Fino ad ora si è visto ciò che accade quando un'istruzione è troppo complessa per completare la fase EX in un solo ciclo. Ma vi sono particolari condizioni che fanno in modo che l'istruzione impieghi più di un ciclo anche per eseguire le altre fasi. Talvolta questi casi sono molto più difficili da considerare poiché i cicli dichiarati per un'istruzione forniscono in realtà un'indicazione del numero dei cicli richiesti per eseguire la fase EX. Altri tipi di stallo delle pipeline sono normalmente provocati dall'ordine nel quale si presentano le istruzioni.

Ad esempio cosa accade alla pipeline nel seguente caso?

```
mov     ax, 1           ; I1
lea     bx, table_1     ; I2
mov     cx, [bx]        ; I3
add     cx, ax          ; I4
```

In alcuni casi possono verificarsi dei blocchi nella generazione degli indirizzi. Ciò che accade è descritto nella Tabella 10.4.

L'istruzione I2 esegue l'operazione LEA nel ciclo 5. Contemporaneamente l'istruzione I3 ha cercato di generare l'indirizzo che essa stessa richiede. Tale indirizzo sarà (DS * 16) + BX. Se questa operazione fosse consentita, l'indirizzo sarebbe errato poiché il registro BX sta per essere aggiornato dall'istruzione precedente, I1. Il 486 e il Pentium rilevano questa condizione e generano un blocco AGI (Address Generation Interlock). Un blocco AGI viene generato quando come componente di un indirizzo viene utilizzato un registro e tale registro è la destinazione dell'istruzione che si trova al ciclo precedente.

10.5 Pipeline accoppiate

Utilizzando il Pentium occorre sempre tenere un considerazione che vi sono due pipeline (la pipe U e la pipe V), un po' come una fabbrica con due catene di montaggio o due nastri trasportatori. Questi due pipeline devono essere costantemente in rapporto l'una con l'altra poiché un'istruzione in una pipeline può modificare i dati in memoria,

i registri e lo stato della CPU. Per questo motivo le attività di una pipeline devono essere note da parte dell'altra pipeline. Solo in questo modo il risultato sarà esattamente lo stesso che si otterrebbe eseguendo sequenzialmente le istruzioni nell'ordine in cui queste vengono lette. Durante l'elaborazione di una pipeline le istruzioni possono entrare in stallo per vari motivi. Per fare in modo che i risultati delle istruzioni vengano prodotti secondo una sequenza corretta, le istruzioni nelle pipe U e V entrano ed escono dalle fasi D1 e D2 contemporaneamente. Quando un'istruzione in una pipe provoca uno stallo, entrano in stallo entrambe le pipeline. Quando un'istruzione nella pipe U entra in stallo nella fase EX, entra in stallo anche l'istruzione nella pipe V.

Tabella 10.3 Funzionamento della pipeline con stallo.

			Fasi			
Cicli	**PF**	**D1**	**D2**	**EX**	**WB**	
1	I1					Inizio I1
2	I2	I1				Inizio I2
3	I3	I2	I1			Inizio I3
4	I4	I3	I2	I1		Inizio I4
5	I5	I4	I3	I2	I1	Inizio I5
6	I6	I5	I4	I3	I2	Inizio I3
7	I6	I5	I4	I3	-	I3 mette in stallo la pipeline
8	I6	I5	I4	I3	-	
9	I6	I5	I4	I3	-	
10	I6	I5	I4	I3	-	Fine I3

Tabella 10.4 Funzionamento pipeline con AGI.

			Fasi			
Cicli	**PF**	**D1**	**D2**	**EX**	**WB**	
1	I1					Inizio I1
2	I2	I1				Inizio I2
3	I3	I2	I1			Inizio I3
4	I4	I3	I2	I1		Inizio I4
5		I4	I3	I2	I1	Fine I1 Esecuzione I2, AGI
6			I3	-	I2	Fine I2 I3 genera l'indirizzo
7			I4	I3	-	

Tabella 10.5 Funzionamento delle pipeline del Pentium.

			Fasi				
Cicli	**Pipe**	**PF**	**D1**	**D2**	**EX**	**WB**	
1	U	I1					Inizio I1
	V	I2					Inizio I2
2	U	I3	I1				Inizio I3
	V	I4	I2				Inizio I4
3	U	I5	I3	I1			Inizio I5
	V	I6	I4	I2			Inizio I6
4	U	I7	I5	I3	I1		Inizio I7
	V	I8	I6	I4	I2		Inizio I8
5	U	I9	I7	I5	I3	I1	Fine I1
	V	I10	I8	I6	I4	I2	Fine I2
6	U	I11	I9	I7	I5	I3	Fine I3
	V	I12	I10	I8	I6	I4	FineI4

Ma se entra in stallo l'istruzione nella pipe V, l'istruzione nella pipe U può procedere alla fase WB. La prossima istruzione (o coppia di istruzioni) non può entrare nella fase EX finché entrambe le istruzioni non saranno passate alla fase WB. Questo evita che l'istruzione nella pipe V entri un stallo nella fase EX e che venga sorpassata dall'istruzione nella pipe U.

Anche se le istruzioni nelle pipeline vengono eseguite in modo indipendente, gli stalli possono essere provocati anche da molti altri fattori oltre che dall'istruzione precedente. Ad esempio:

```
mov   bx, offset mem  ; I1
mov   ax, 1           ; I2
mov   cx, 1000        ; I3
add   dx, [bx]        ; I4
```

In questo codice descritto dalla Tabella 10.4, l'operazione di MOV in BX provoca un blocco AGI per l'istruzione ADD che si trova tre istruzioni più avanti.

10.6 Ritardi nelle pipeline del 486

Nel 486 vi possono essere due ritardi nelle pipeline:

- ritardi nella generazione dell'indirizzo (solo 486);
- blocco di generazione degli indirizzi (AGI).

Sul 486, quando il calcolo di un indirizzo effettivo utilizza un registro base e un registro indice, la fase D2 della pipeline richiede un ciclo in più. È richiesto un ciclo aggiuntivo anche quando l'istruzione contiene un valore di scostamento e un valore immediato (una costante).

Tabella 10.6 Funzionamento delle pipeline nel Pentium con un blocco AGI.

			Fasi				
Cicli	**Pipe**	**PF**	**D1**	**D2**	**EX**	**WB**	
1	U	I1					Inizio I1
	V	I2					Inizio I2
2	U	I3	I1				Inizio I3
	V	I4	I2				Inizio I4
3	U	I5	I3	I1			
	V	I6	I4	I2			
4	U		I5	I3	I1		I3 mette in stallo I4
	V		I6	I4	I2		I4 AGI
5	U		I5	I3	-	I1	Fine I1
	V		I6	I4	-	I2	Fine I2
6	U			I5	I3	-	
	V			I6	I4	-	
7	U				I5	I3	Fine I3
	V				I6	I4	Fine I4

Sul 486 si verifica un blocco AGI quando un'istruzione scrive su un registro che viene utilizzato nel calcolo di un indirizzo effettivo per la prossima istruzione.

10.7 Ritardi nelle pipeline del Pentium

Nel Pentium vi sono quattro tipi di ritardi che non influenzano direttamente l'accoppiabilità delle istruzioni ma introducono cicli aggiuntivi e devono pertanto essere considerate quando si devono riorganizzare le istruzioni. Tali ritardi sono dovuti a:

- conflitti di banco nella memoria cache;
- blocchi AGI;
- ritardi per il byte di prefisso;
- ritardi di sequenza.

Conflitti di banco nella memoria cache

Quando due istruzioni accoppiate accedono allo stesso banco di memoria cache, nella seconda istruzione viene introdotto un ritardo di un ciclo. Un conflitto di banco si verifica quando nei due indirizzi fisici i bit da 2 a 4 sono uguali. Talvolta si tratta di un problema difficile da risolvere, specialmente nel caso di subroutine di basso livello che ricevono solamente un puntatore a un dato. La strategia migliore consiste nel tentare di non accoppiare le istruzioni che possono accedere allo stesso banco di memoria cache.

Quello del conflitto di banco è un concetto estremamente importante e dunque è fondamentale comprenderne bene le cause. Osservando attentamente l'aspetto della memoria cache illustrato nella Figura 10.1, si può osservare che ogni linea è composta da un indirizzo a 32 bit (vengono mostrati solo 16 bit) che può essere suddiviso in tre parti:

```
bit:   0-1    byte nella dword
bit:   2-4    banco di cache
bit:   5-31   indirizzo della linea della cache
```

Le linee della cache vengono sempre riempite sui limiti di blocchi da 32 byte di memoria e dunque è facile determinare quando si verifica un conflitto di banco. La situazione si verifica quando il numero del banco coincide (ovvero quando i bit da 2 a 4 negli indirizzi sono uguali). Ecco un esempio:

```
lbl:
mov    al, [si]
mov    bl, [si+1]
...
add    si, 2
loop   lbl
```

Questo ciclo di istruzioni provoca un conflitto di banco ad ogni iterazione nel caso in cui SI sia inizializzato con un numero pari. Quando invece SI parte da un numero dispari, il ciclo presenterà un conflitto di banco il 50% delle volte. Dunque può acca-

Banco:	0	1	2	3	4	5	6	7
Byte:	0-3	4-7	8-B	C-F	10-13	14-17	18-1B	1C-1F
Indirizzo della linea di cache (bit da 5 a 15)								

0. 0000 0000 000
1. 0000 0000 001
2. 0000 0000 011
3. 0000 0000 010

...

...

127.

```
Questo è solo un esempio. Vi son
o quattro righe di testo ma occo
rdine sequenziale.
rre notare che non appaiono in o
```

Note: ogni riga contiene 32 byte partendo dal confine di un blocco da 32 byte nella memoria principale. L'indirizzo principale di un banco di memoria cache è un indirizzo fisico a 32 bit con i cinque bit inferiori uguali a zero. La cache è dotata di 128 banchi che possono essere disposti in qualsiasi ordine. Non necessariamente banchi di cache contigui devono contenere blocchi di memoria contigui. Si noti che le righe 2 e 3 della cache contengono indirizzi di memoria non coordinati.

Figura 10.1 Aspetto della memoria cache.

dere che dati mal allineati aumentino la velocità del programma! Questo fenomeno verrà discusso più in dettaglio nel Capitolo 13.

Blocchi AGI (Address Generation Interlock)

Si è già parlato dei ritardi AGI nella sezione precedente. La regola generale è che un ritardo AGI si verifica quando un'istruzione in un determinato ciclo scrive su un registro che viene utilizzato nel calcolo di un indirizzo effettivo da un'istruzione che verrà eseguita al prossimo ciclo. Questo può verificarsi in due casi: quando un'istruzione in un ciclo modifica un registro che costituisce la porzione base o indice nel calcolo di un indirizzo effettivo per il ciclo successivo o quando un'istruzione in un ciclo modifica SP (o ESP) e l'istruzione successiva utilizza tale valore. Tuttavia vi è un'eccezione alla regola AGI. Quando le istruzioni utilizzano entrambe SP (o ESP) implicitamente, come nel caso delle istruzioni PUSH o POP, non vi sarà alcun blocco AGI. Ecco alcuni esempi di blocchi AGI:

```
inc   bx            ; due INC
inc   ax
mov   cx, [si]      ; due MOV
mov   dx, [bx]      ; ritardo AGI poiché BX viene modificato nel ciclo precedente
pop   bx            ; due POP
pop   ax
ret                 ; nessun ritardo AGI
pop   bx            ; due POP
pop   ax            ;
add   sp, 10
ret                 ; ritardo AGI
```

Ritardo per il byte del prefisso

Il terzo ritardo è dovuto alla lettura del byte del prefisso. Sul 486 i byte del prefisso non aggiungono alcun ciclo. Sul Pentium il prefisso, ad esempio quello che consente di superare i limiti del segmento, richiede un ciclo in più. Inoltre occorre ricordare la regola 6 (Tabella 9.2) poiché un'istruzione con prefisso non può essere accoppiata nella pipe V. Anche se questo non sempre è vero, è bene considerare sempre che i byte di prefisso sono istruzioni non accoppiabili. Questo modello concettuale funziona bene per prevedere il numero di cicli richiesti per un blocco di istruzioni (per un elenco dei prefissi si consulti l'Appendice F). Nell'esempio successivo viene presentato un prefisso che impedisce l'accoppiamento di due istruzioni che altrimenti potrebbero essere accoppiate:

```
lbl:
        mov   al, [si]        ; 1 ciclo
        mov   bl, ES: [di]    ; 2 cicli (1 per il prefisso, 1 per MOV)
...
loop lbl
```

Riordinando le istruzioni è possibile disporre l'istruzione con il prefisso (in questo caso l'uscita dal segmento) in modo da consentirne l'accoppiamento nella pipe U:

```
lbl:
        mov     bl, ES:[di]     ; 2 cicli
        mov     al, [si]        ; 0 cicli
        loop    lbl
```

Ritardo di sequenza

Il quarto ritardo è dovuto alla sequenza di esecuzione delle istruzioni. La maggior parte delle istruzioni semplici viene eseguita in un ciclo di CPU poiché tali istruzioni vengono interpretate direttamente (non sono implementate tramite microcodice). Ma vi sono forme di istruzioni aritmetico-logiche che richiedono 2 o 3 cicli. Ad esempio:

```
add     mem, reg        ; 3 cicli (leggi-modifica-scrivi)
add     reg, mem        ; 2 cicli (leggi-modifica-registro)
```

L'hardware di controllo delle sequenze consente di interpretare queste istruzioni come istruzioni semplici. Dunque sono accoppiabili sia la forma a 2 cicli che la forma a 3 cicli. Tuttavia, quando vengono accoppiate due istruzioni con operazioni "leggi-modifica-scrivi" (forme a 3 cicli) vi sarà un ritardo di due cicli. Ad esempio:

```
add     ax, [bx]        ; 2 (le forme a due cicli si sovrappongono)
add     cx, [si]        ; 0

add     [bx], 2         ; 3
add     [si], 2         ; 2 (il primo ciclo si sovrappone al precedente)
```

Le ultime due istruzioni richiedono 3 cicli l'una. Se vengono accoppiate richiedono un totale di cinque cicli a causa del ritardo di due cicli introdotto dalla sequenza.

Eliminazione del ritardo di sequenza

L'obiettivo è quello di eliminare il ritardo di due cicli. Utilizzando un registro aggiuntivo è possibile riscrivere il codice nel seguente modo:

```
mov     ax, [bx]        ; 1
add     ax, 2           ; 1
mov     [bx], ax        ; 1
mov     ax, [si]        ; 0
add     ax, 2           ; 1
mov     [si], ax        ; 1
```

Ma anche questa forma richiede cinque cicli (si dovrebbero accoppiare la terza e la quarta istruzione). Questo porta a credere che quello illustrato sia esattamente il modo in cui la CPU pone in sequenza le operazioni, ovvero utilizzando un registro interno aggiuntivo. Dunque, ora che si è scoperto questo "registro aggiuntivo" se ne può tenere conto e riscrivere il codice:

```
mov   ax, [bx]     ; 1
add   ax, 2        ; 1
mov   [bx], ax     ; 1
add   [si], 2      ; 2 (il primo ciclo si sovrappone al precedente)
```

Anche questa versione richiede cinque cicli ma è più compatta. Il problema è che scrivendo il codice in questo modo non è possibile trovare altre opportunità per l'accoppiamento delle istruzioni. Dunque si può provare a riscrivere il codice utilizzando due registri aggiuntivi:

```
mov   ax, [bx]     ; 1
add   ax, 2        ; 1
mov   [bx], ax     ; 1
mov   cx, [si]     ; 0
add   cx, 2        ; 1
mov   [si], cx     ; 1
```

Anche questa forma richiede cinque cicli ma può essere riordinata nel seguente modo:

```
mov   ax, [bx]     ; 1
mov   cx, [si]     ; 0
add   ax, 2        ; 1
add   cx, 2        ; 0
mov   [bx], ax     ; 1
mov   [si], cx     ; 0
```

Questa sequenza richiede solo tre cicli ed è scritta secondo un classico stile "carica-modifica-salva". Questo è il motivo per il quale uno dei concetti di base delle macchine RISC, l'architettura "carica/salva" è così importante. Scrivendo il codice secondo questo stile è più facile riordinare le istruzioni per ottenere un'esecuzione più efficiente.

Se si devono salvare e ripristinare i due registri utilizzati per modificare questa sequenza di codice, le due operazioni richiederanno esattamente i due cicli risparmiati. Ma se le istruzioni PUSH e POP sono all'esterno di un ciclo di istruzioni, si sarà ottenuta una riduzione del codice del ciclo di istruzioni da 5 a 3 cicli di CPU, un miglioramento del 40%.

10.8 La pipeline in virgola mobile del Pentium

Questo capitolo si conclude con la descrizione della pipeline e delle istruzioni in virgola mobile del Pentium. In ogni caso si tratta di un argomento non necessario per la comprensione della maggior parte del contenuto di questo manuale. Solo nel Capitolo 16 si troveranno infatti esempi di programmazione in virgola mobile. Per comprendere appieno il funzionamento delle operazioni in virgola mobile è preferibile avere conoscenze di programmazione in virgola mobile con il coprocessore 8087.

La pipeline per operazioni in virgola mobile prevede otto fasi, di cui le prime cinque sono uguali a quelle dell'unità intera. Ecco una breve descrizione di ciò che avviene durante ogni fase della pipeline:

PF: Le istruzioni vengono lette dalla cache o dalla memoria e vengono inserite nella coda di prefetch.

D1: Le istruzioni vengono decodificate e suddivise un componenti, ovvero codice operativo e operandi. Se l'istruzione contiene un prefisso, è necessario un ciclo aggiuntivo.

D2: Se presente viene calcolato l'indirizzo effettivo dell'operando in memoria. Sul 486 è necessario un ciclo aggiuntivo se l'indirizzo contiene sia la componente base che l'indice oppure sia lo scostamento che un valore costante.

EX; Conversione dei dati nel formato in virgola mobile.

X1: Conversione dei dati in virgola mobile nel formato interno.

X2: Seconda fase dell'esecuzione.

WF: Esecuzione degli arrotondamenti e scrittura dei risultati in virgola mobile in un registro.

ER: Aggiornamento della word di stato.

Le istruzioni in virgola mobile non possono essere accoppiate con istruzioni intere ma alcune istruzioni in virgola mobile possono essere accoppiate fra loro. Le istruzioni intere e in virgola mobile possono essere eseguite contemporaneamente (a tale proposito si veda la discussione seguente). Le regole di accoppiamento delle istruzioni in virgola mobile sono molto rigide; a tale proposito si consulti la Tabella 10.7.

La famiglia dei coprocessori matematici 8087 ha un'architettura a stack, con un funzionamento simile alle calcolatrici scientifiche Hewlett Packard. Lo stack è costituito da 8 registri da 80 bit e un puntatore allo stack. Normalmente, non è importante sapere in quale registro si trova una variabile; è significativa solo la sua posizione rispetto al puntatore allo stack. I registri in virgola mobile fanno dunque sempre riferimento alla cima dello stack:

```
st      cima dello stack (ultimo elemento caricato)
st(0)   uguale a st
st(1)   penultimo elemento caricato
st(2)   terzultimo elemento caricato
...
st(7)   fine stack
```

Gli operandi possono essere caricati nello stack utilizzando ad esempio l'istruzione FLD. I risultati possono essere copiati in memoria dallo stack tramite l'istruzione FST e le sue varianti. Alla conclusione delle istruzioni, i risultati possono anche essere scaricati dallo stack. Il codice mnemonico di queste istruzioni termina con la lettera P. Ecco un breve esempio:

```
                                      st(0)   st(1)  st(2)
fld     n0      ; carica n0           n0      -      -
fld     n1      ; carica n1           n1      n0     -
fadd            ; st(0) = st(0)+st(1) n1+n0   n0     -
fstp    ans_1   ; ans_1 = st(0), pop  n0      -      -
fld     n2      ; carica n2           n2      n0
fld     n3      ; carica n3           n3      n2     n0
fmul            ; st(0) = st(0)*st(1) n3*n1   n1     n0
fst     ans_2   ans_2 = st(0)         n3*n1   n1     n0
```

Tutte le istruzioni a due operandi richiedono che un operando di origine si trovi sulla cima dello stack ovvero in st(0). Anche molte delle istruzione più comuni hanno come destinazione la cima dello stack. Questo provoca un "collo di bottiglia" in cima allo stack. Come si può vedere all'esempio precedente, ogni istruzione usa infatti la cima dello stack, ovvero st(0). Questo collo di bottiglia può essere eliminato utilizzando l'istruzione FXCH che scambia rapidamente il registro st(0) con uno degli altri registri. Sul Pentium, l'istruzione FXCH richiede 0 cicli poiché può essere accoppiata con una qualsiasi delle istruzioni contenute nella Tabella 10.7.

Ritardi nella pipeline in virgola mobile

Le prestazioni del sistema possono degradare per vari motivi. Talvolta è difficile rilevare questa riduzione di prestazioni ma altre volte un'istruzione può anche richiedere un numero di cicli triplo. I ritardi che possono verificarsi sono:

- latenza di scrittura;
- ritardo di FST
- ritardo di ripetizione di FMUL
- ritardo FXCH-istruzione intera

Le istruzioni in virgola mobile, come FADD e FMUL, sono state ottimizzate sul Pentium in modo da richiedere un solo ciclo di CPU. Tuttavia, a causa della complessità della pipeline in virgola mobile, se il risultato di un'operazione è richiesto come input dall'istruzione successiva si verifica uno stallo. La latenza di scrittura può costare quattro cicli aggiuntivi. L'unico modo per risolvere questo ritardo consiste nell'utilizzare in modo intervallato altre istruzioni in virgola mobile che non generino conflitti. La Tabella 10.8 elenca tutte le istruzioni influenzate. Il Capitolo 16 contiene alcuni esempi che mostrano come è possibile eliminare questi ritardi.

Il ritardo FST è un ritardo di un ciclo che si aggiunge alla latenza di scrittura quando un'istruzione FST (Floating-point STore) utilizza il risultato di una precedente operazione in virgola mobile.

Quando l'istruzione FMUL è seguita immediatamente da un'altra istruzione FMUL, la massima velocità scende a 2 cicli invece di 1 a causa della doppia fase di esecuzione che provoca un conflitto.

Per ottenere le massime prestazioni, l'istruzione FXCH deve essere seguita da un'altra istruzione in virgola mobile e non da un'istruzione intera. Quando l'istruzione FXCH è seguita da un'istruzione intera, si verifica un ritardo. Tale ritardo è di un ciclo per istruzioni accoppiate e sicure e di quattro cicli per istruzioni accoppiate ma non sicure. Allora ci si potrebbe chiedere quando un'istruzione è sicura?

Come riconoscere le istruzioni sicure

Su carta ogni istruzione matematica è sicura poiché è stato definito un simbolo per ogni situazione. Vi sono simboli matematici per l'infinito (positivo e negativo); è teoricamente possibile scrivere qualsiasi numero. Purtroppo un computer ha invece dei limiti; ad esempio un numero in virgola mobile in precisione semplice deve essere compreso fra 1.18×10^{-38} e 3.4×10^{38} oppure un valore negativo nello stesso interval-

Tabella 10.7 Regole di accoppiamento delle istruzioni in virgola mobile.

1. L'istruzione nella pipe U deve essere un'istruzione in virgola mobile semplice.	
2. L'istruzione nella pipe V deve essere FXCH.	
Istruzioni in virgola mobile semplici:	
FABS	ABSolute value
FADD	ADD
FADDP	ADD and Pop
FCHS	CHange Sign
FCOM	COMpare real
FCOMP	COMpare real and pop
FDIV	DIVide
FDIVP	DIVide and Pop
FDIVR	DIVide Reverse
FDIVRP	DIVide Reverse and Pop
FLD	LoaD reale, in precisione singola, doppia o st(i)
FMUL	MULtiply
FMULP	MULtiply and Pop
FSUB	SUBtract
FSUBP	SUBtract and Pop
FSUBR	SUBtract Reverse
FSUBRP	SUBtract Reverse and pop
FTST	TeST
FUCOM	Unordered COMpare real
FUCOMP	Unordered COMpare real and Pop
FUCOMPP	Unordered COMpare real and Pop (due volte)

lo o infine lo 0. Un'operazione che fornisca un risultato all'esterno di questi limiti provoca un errore del programma. Un'eccezione in virgola mobile è un errore numerico come ad esempio una divisione per 0, un underflow o un overflow. Solo alcune istruzioni possono generare questo tipo di errori (ad esempio l'errore di divisione per 0 può essere ottenuto solo tramite una divisione).

Tabella 10.8 Cicli delle istruzioni in virgola mobile con latenze di scrittura.

Istruzioni	Cicli	Con il ritardo di latenza
FADD, FADDP	1	3
FMUL, FMULP	1	3
FSUB, FSUBP, FSUBR, FSUBRP	1	3
FCOM, FTST	1	4
FUCOM, FUCOMP, FUCOMPP	1	4
FSTSW AX	2	6
FSTSW	2	5
FICOM	4	8
FIADD, FISUB, FILD	4	7

Inoltre, ognuna delle eccezioni può essere disattivata sotto il controllo del programmatore che può chiedere all'unità per calcoli in virgola mobile di utilizzare una routine di correzione per questa operazione. Ad esempio, se una divisione fornisce un numero molto piccolo, troppo piccolo per essere rappresentato nei valori consentiti dall'unità in virgola mobile, si verifica un errore di underflow. Se si mascherano gli underflow, l'unità in virgola mobile sostituirà un risultato "speciale" pari a 0.0 e non genererà l'eccezione. La generazione di risultati speciali, quando viene mascherata una condizione di eccezione, viene eseguita tramite l'impiego del microcodice contenuto nell'unità in virgola mobile.

Un'istruzione è considerata sicura se non può generare un'eccezione in virgola mobile e se non usa il microcodice dell'unità in virgola mobile per generare risultati particolari. Le possibilità delle eccezioni sono enormi ma ecco come funzionano per le istruzioni FADD, FSUB, FMUL e FDIV. Gli esponenti di queste istruzioni devono essere compresi nei seguenti limiti:

```
-8190 <= esponente <= 8190
```

Come si può notare questi limiti sono molto superiori rispetto ai numeri in precisione semplice o doppia poiché l'unità in virgola mobile converte tutti gli operandi nel formato interno a 80 bit.

Quando un'istruzione è dichiarata sicura, l'istruzione successiva può completare la fase EX della pipeline. Quando un'istruzione è dichiarata non sicura, l'istruzione in virgola mobile successiva entra in stallo nella fase EX fino al termine (senza eventuali eccezioni) dell'istruzione non sicura. Osservando la descrizione della pipeline in virgola mobile, si può vedere che lo stallo dura almeno quattro cicli. Si noti che questo stallo si verifica anche se il funzionamento dell'unità in virgola mobile non genera un'eccezione. Un'istruzione dichiarata come non sicura infatti non sempre provoca un'eccezione; pertanto l'unità in virgola mobile deve attendere che l'istruzione abbia termine per vedere se si verifica un'eccezione per poi far procedere l'istruzione successiva.

Elaborazione concorrente di istruzioni intere e in virgola mobile

Per il fatto che l'unità intera e l'unità in virgola mobile sono distinte, è possibile che le istruzioni in virgola mobile vengano eseguite in parallelo alle istruzioni intere. Questo era possibile fin dall'8086/8087. Poiché in generale le istruzioni in virgola mobile vengono eseguite in un tempo più lungo, l'unità interna può eseguire più istruzioni prima che l'unità in virgola mobile possa completare una singola istruzione. Ad esempio, questa porzione di codice calcola la radice quadrata di un array di numeri in precisione semplice:

```
lbl:
        fld     dword ptr [esi]     ; carica un reale di 4 byte
        fsqrt                       ; calcola la radice quadrata
        fst     dword ptr [edi]     ; memorizza il risultato
        add     esi, 4              ; fa avanzare il puntatore di origine
        add     edi, 4              ; fa avanzare il puntatore di destinazione
        dec     ecx                 ; decrementa il contatore del ciclo
        jnz     lbl                 ; continua il ciclo se ecs è diverso da 0
```

Il ciclo successivo esegue la stessa operazione ma sfrutta il lungo tempo di esecuzione dell'istruzione **FSQRT** per eseguire alcune istruzioni interne necessarie per la gestione del ciclo:

```
lbl:
        fld     dword ptr [esi]     ; carica un reale di 4 byte
        fsqrt                       ; calcola la radice quadrata
        add     esi, 4              ;; fa avanzare i puntatori

        add     edi, 4              ;; durante l'esecuzione della radice quadrata
        dec     ecx                 ;; decrementa il contatore del ciclo
        fwait                       ; attende il termine del calcolo della radice quadrata
        fst     dword ptr [edi-4]   ; memorizza il risultato
        jnz     lbl                 ; continua il ciclo se ecs è diverso da 0
```

Quando si esegue questo codice si può osservare che l'esecuzione delle tre istruzioni che seguono **FSQRT** richiede zero cicli.

Poiché il Pentium può eseguire molte istruzioni in virgola mobile in un solo ciclo, l'esecuzione concorrente diventa una strategia meno importante. Tuttavia le funzioni trigonometriche, logaritmiche e così via possono ancora richiedere più di 100 cicli e la divisione richiede 39 cicli, dunque vi è ancora l'opportunità di sfruttare le possibilità di elaborazione concorrente.

La gestione dell'esecuzione parallela nelle unità intera e in virgola mobile può essere difficile, specialmente quando un programma deve tenere conto delle eccezioni numeriche. I linguaggi di alto livello tendono a eseguire queste operazioni in modo affidabile ma lento, principalmente perché le operazioni in virgola mobile non sono normalmente contenute in una libreria che possa essere isolata dal codice di controllo del ciclo generato dal compilatore. Nel Capitolo 16 verranno esaminati alcuni esempi di accoppiamento di istruzioni intere, di esecuzione concorrente di istruzioni intere e in virgola mobile e di accoppiamento di istruzioni in virgola mobile per aumentare le prestazioni dell'unità in virgola mobile anche più di dieci volte.

Capitolo 11

Uso del programma di ottimizzazione per il Pentium

Questo capitolo e il Capitolo 12 descrivono l'utilizzo e il funzionamento di due strumenti forniti con il disco allegato. Questo capitolo descrive il programma PENTOPT, un analizzatore di ottimizzazione per il Pentium. Il prossimo capitolo descrive una libreria di procedure che consentono di misurare le prestazioni delle sezioni più critiche del codice.

La versione di PENTOPT fornita è perfettamente funzionale e si basa sulla versione commerciale PENTOPT Professional. Entrambi i programmi derivano poi da un altro prodotto, ASMFLOW Professional, un generatore di flow chart di utilizzo generale per il linguaggio assembler e un programma di analisi del codice sorgente.

In pratica, PENTOPT crea un flow chart del codice sorgente assembler e produce un'analisi di ottimizzazione per il Pentium rispetto ad ogni istruzione. La versione commerciale di PENTOPT è dotata di alcune funzionalità aggiuntive che consentono di gestire progetti più estesi costituiti da più file. La versione di PENTOPT fornita nel disco è però più che adeguata per tutti gli esempi presentati in questo manuale e per esempi di analoga complessità.

Ecco un esempio dell'output presentato da PENTOPT.

```
12      main proc               ; Pentium cycles
13      push   bx               ;     1 cy     UV   1
14      push   cx               ;     1 cy     UV   *
15      push   dx               ;     1 cy     UV   *
17      mov    bx, offset tbll  ;     1 cy     UV   *
18      mov    cx, [bx]         ;     1 cy     Uv   4 AGI-1
19      shl    cx, 1            ;     1 cy     U    2
20      mov    bx, offset array ;     1 cy     UV   *
22      cmp    cx, 0            ;     1 cy     UV   *
```

```
23          je     main_3        ;    1 cy          V    *
24   main_1:
25          mov    ax, [si]      ;    1 cy          UV   2
26          mov    bx, [si+2]    ;    1 cy          UV   *
27          mul    bx            ;    10 cy/11 cy   NP
29          cmp    ax, 1         ;    1 cy          UV   1
30          je     main_2        ;    1 cy          V    *
32          add    si, 4         ;    1 cy          UV   2
33          cmp    si, x_end     ;    2 cy          UV   4
34          jae    main_3        ;    1 cy          V    *
35          dec    cx            ;    1 cy          UV   2
36          jnz    main_1        ;    1 cy          V    *
37   main_2:
38          inc    si            ;    1 cy          UV   2
39          inc    si            ;    1 cy          UV   3
40   main_3:
41          pop    dx            ;    1 cy          UV   *
42          pop    cx            ;    1 cy          UV   *
43          pop    bx            ;    1 cy          UV   *
44          ret                  ;    2 cy          NP
46          main endp

48          proc2  proc near     ; Pentium cicli
50          push   bx            ;    1 cy          UV   1
51          push   cx            ;    1 cy          UV   *
52          push   dx            ;    1 cy          UV   *
54   p0:
55          inc    di            ;    1 cy          IN   *
56   p1:
57          inc    dx            ;    1 cy          IN   *
58   p2:
59          add    si, 2         ;    1 cy          UV   *
60          mov    ax, [si]           1 cy          UV   4 AGI-1
61          aul    ax            ;    10 cy/11 cy   NP
62          div    bx            ;    17 cy/41 cy   NP
64          dec    cx            ;    1 cy          UV   1
65          jnz    p2            ;    1 cy          V    *
67          mul    ax            ;    10 cy/11 cy   NP
68          div    bx            ;    17 cy/41 cy   NP
69          dec    bx            ;    1 cy          IN   1
70          jnz    pl            ;    1 cy          V    *
72          add    ax, bx        ;    1 cy          UV   2
73          jnc    p0            ;    1 cy          V    *
75          pop    dx            ;    1 cy          UV   2
76          pop    cx            ;    1 cy          UV   *
77          pop    bx            ;    1 cy          UV   *
78          ret                  ;    2 cy          NP
80          proc2  endp
```

Note: * = accoppiabile all'istruzione precedente.
1 = l'istruzione precedente non è accoppiabile (NP)
2 = mancata corrispondenza delle pipe U e V (la precedente non è U e questa non è V)

3 = destinazione ripetuta nei registri (scrittura/scrittura)
4 = conflitto nei registri (scrittura/lettura)
5 = conflitto nel puntatore allo stack o nel registro dei flag (scrittura/lettura)
6 = conflitto nell'operando in memoria (scrittura/scrittura o scrittura/lettura)
AGI*n* = blocco AGI sull'istruzione *n*

L'esecuzione di PENTOPT è semplice: occorre infatti solo specificare il nome di un file assembler nella riga di comando:

```
C:> pentopt esempio.asm
```

In questo caso l'output viene inviato direttamente sullo schermo. Per inviare l'output su un file o sulla stampante, si può utilizzare la redirezione dell'output del DOS:

```
C:> pentopt esempio.asm > output.txt
```

11.1 Funzionamento del programma

Il funzionamento interno di PENTOPT è piuttosto complesso; tuttavia è possibile esporre le funzionalità di base. PENTOPT esamina ogni istruzione del codice assembler e determina se si tratta di un'istruzione per la CPU o meno. In caso affermativo, PENTOPT determina la categoria dell'istruzione:

NP	non accoppiabile;
UV	accoppiabile nelle pipe U o V ;
U	accoppiabile solo nella pipe U;
V	accoppiabile solo nella pipe V.

Questa categoria è indicata nella colonna delle note nella parte destra dello schermo. Quando un'istruzione è accoppiabile nella pipe V, PENTOPT cerca di determinare se l'istruzione può essere accoppiata all'istruzione precedente. In caso affermativo, a destra della categoria viene visualizzato un asterisco. Ma l'asterisco NON significa che l'istruzione verrà effettivamente accoppiata nella pipe, ma semplicemente che questo è possibile. L'assenza dell'asterisco significa che l'istruzione non può essere accoppiata. Se manca l'asterisco e l'istruzione potrebbe essere accoppiata nella pipe V, può essersi verificata una delle condizioni indicate al termine della Figura 11.1.

Ecco due motivi per i quali due istruzioni potrebbero non essere accoppiabili anche quando l'operazione fosse possibile (cioè viene visualizzato l'asterisco):

- l'istruzione precedente è accoppiabile e accoppiata nella pipe V;
- l'istruzione può essere la prima istruzione dopo un salto o una chiamata.

11.2 I blocchi AGI (Address Generation Interlock)

Oltre alle informazioni di accoppiamento, vengono rilevati i blocchi AGI, indicati a destra delle informazioni di accoppiamento (per una descrizione dei blocchi AGI, si

consulti il Capitolo 10). Le informazioni AGI includono il numero di riga dell'istruzione che genera l'indirizzo che provoca il conflitto con l'istruzione corrente. Al momento dell'esecuzione, è anche possibile che un blocco AGI non si verifichi anche quando tale blocco viene indicato per un'istruzione che presiede di due o tre posizioni l'istruzione indicata.

Possono verificarsi blocchi AGI anche sul 486. Tuttavia tali blocchi si verificano solo quando l'indirizzo viene generato dall'istruzione precedente. Pertanto, quando si scrive codice destinato solo al 486 si sarà interessati solo al blocco AGI-1.

Capitolo 12

Valutazione delle prestazioni con un timer software

Fino ad ora sono state fatte alcune assunzioni rispetto al numero dei cicli di CPU e alle possibilità di accoppiamento delle istruzioni. Ma tali assunzioni devono essere verificate sul campo misurando le effettive prestazioni del codice. Il Pentium è dotato di 8 KB di memoria cache per il codice e dunque praticamente ogni ciclo può essere eseguito all'interno della cache risultando altamente ottimizzato. Ma è comunque necessario valutare le prestazioni del codice per verificare che operi nel modo atteso.

Vi sono almeno tre metodi per valutare le prestazioni del codice su un Pentium:

- utilizzare un dispositivo hardware come un ICE;
- utilizzare il timer interno del Pentium;
- utilizzare un timer software per controllare i chip di temporizzazione del PC.

12.1 Gli emulatori ICE (In Circuit Emulator)

La maggior parte dei programmatori non dispone di un ICE (In Circuit Emulator) e dunque se ne parlerà solo in termini generali. Un ICE è in grado di controllare e memorizzare ogni ciclo macchina in un buffer di memoria RAM ad alta velocità. Il numero di cicli che è possibile osservare quando viene raggiunto un breakpoint dipende dalle dimensioni del buffer. Il grande vantaggio consiste nella possibilità di conoscere esattamente ciò che accade ad ogni ciclo e, sempre che il buffer sia sufficientemente esteso, nella possibilità di tener conto di ogni singolo ciclo di un loop. Lo svantaggio è il prezzo di questi apparecchi che in genere costano molto più di un sistema medio. Inoltre sarà necessario utilizzare nuovo hardware per ogni modello di CPU.

12.2 Il timer interno del Pentium

Il Pentium è dotato di un proprio timer interno. Tale microprocessore è dotato di una nuova istruzione (RDTSC) che non è ben documentata da Intel. In particolare il manuale "*The Intel Pentium Processor User's Manual, Volume 3: Architecture and Programming Manual*" non la cita se non nell'Appendice A (nell'elenco dei codici operativi) e nel Capitolo 10 (nella descrizione del bit TSD - Time Stamp Disable). In particolare manca ogni descrizione del significato o dell'utilizzo di tale istruzione.

Ecco come funziona il timer del Pentium. Il Pentium è dotato di un contatore interno a 64 bit che viene incrementato ad ogni ciclo. Questo significa che il timer ha una precisione di un ciclo macchina ogni 8800 anni (a 66 MHz). Il MASM 6.11 è in grado di assemblare l'istruzione RDTSC mentre in altri casi sarà necessario specificare direttamente il codice operativo dell'istruzione:

```
rdtsc                   ; ReaD Time Stamp Counter
db 0Fh, 31h             ; codice operativo equivalente a rdtsc
```

L'operazione eseguita da RDTSC è piuttosto semplice: restituisce il valore del contatore interno dei cicli nei registri EDX:EAX (EDX contiene i 32 bit più alti ed EAX contiene i 32 bit più bassi).

Ecco dunque una routine che utilizza questa nuova istruzione:

```
rdtsc
mov   start_low, eax
mov   start_hi, edx
call  test_proc
rdtsc
sub   eax, start_low
sbb   edx, start_hi
```

L'istruzione RDTSC può essere utilizzata in qualsiasi programma a 16 bit operante in modalità reale. In modalità protetta tale istruzione può (in alcuni casi) essere protetta dal sistema operativo. In questo caso l'istruzione TSD richiede un livello di protezione pari a 0. Questo significa che una normale applicazione (operante al livello 3) può generare una violazione di protezione. I programmi in modalità protetta possono però eseguire il test di un bit che consente di determinare se l'istruzione RDTSC è privilegiata o meno. Quando il bit 2 di CR4 è uguale a 1, RDTSC è un'istruzione privilegiata. Una completa descrizione dei registri speciali non rientra negli scopi di questo manuale. In pratica, a partire dal 386, vi sono numerosi registri di controllo (CR*x*), registri di debug (DR*x*) e registri di test (TR*x*). La scrittura di tali registri di controllo è considerata un'operazione privilegiata mentre qualsiasi applicazione può tranquillamente leggere il contenuto di tali registri. Per verificare se il l'istruzione RDTSC è privilegiata basta utilizzare le seguenti istruzioni:

```
mov   eax, CR4
test  eax, 4
jz    ok
priv:          ; RDTSC privilegiata
ok:
               ; RDTSC available
```

Anche se RDTSC è un'istruzione privilegiata, questo non significa che non possa essere eseguita in un'applicazione operante in modalità protetta. Quando il sistema operativo acquisisce il controllo per un errore di protezione generale, può decidere di eseguire l'istruzione e restituire i risultati all'applicazione come se non si trattasse di un'istruzione privilegiata. Per questo è consigliabile utilizzare RDTSC in modo non sia necessario leggere il valore due volte.

12.3 Un timer software

Lo svantaggio dell'istruzione RDTSC è il fatto che rende il codice specifico per il Pentium. Lo sviluppo di questo software di valutazione è iniziato prima che fosse effettivamente disponibile il chip Pentium. Dopo aver appreso della presenza dell'istruzione RDTSC, si pensava di abbandonare questo metodo di valutazione. Tuttavia, una riflessione più approfondita ha portato a pensare che poteva essere uno svantaggio creare codice specifico per il Pentium. Qualcuno poteva non essere ancora in possesso di un Pentium o poteva averne un accesso limitato. Grazie a un timer software, era possibile eseguire lo stesso codice anche su altre macchine, come ad esempio un 486. In ogni caso la scelta è sempre nelle mani del programmatore che può scegliere lo strumento più adatto a seconda delle situazioni.

Il timer fornito nel disco allegato consente di verificare i risultati degli esempi contenuti in questo manuale. Questo timer presenta numerosi vantaggi. Innanzitutto consente di scrivere e verificare i programmi su qualsiasi macchina. In secondo luogo consente di eseguire lo stesso codice su più macchine per confrontare le prestazioni su un 386, un 486 o un Pentium.

Tuttavia quando si valutano le prestazioni del codice sul Pentium occorre fare particolare attenzione poiché è fondamentale l'integrazione della cache del codice con le tecniche di accoppiamento delle istruzioni. La maggior parte delle istruzioni accoppiabili può essere effettivamente accoppiata solo a partire dalla seconda esecuzione che avviene dalla memoria cache. Questo significa che il seguente test non rappresenta l'effettiva velocità di esecuzione del codice:

```
call    timer_on
rept    1000

mov     bx, 1           ; 1 ciclo
mov     ax, 1           ; 1 ciclo, 0 se accoppiata
endm
call    timer_off
```

Questo test fornisce un risultato di 2000 cicli. Sarebbe invece più significativo utilizzare il seguente test:

```
        call test1      ; precaricamento della cache con il codice e i dati
        call timer_on   ; avvio del timer
        call test1      ; esecuzione del test
        call timer_off  ; arresto del timer
test1 proc
```

```
        rept 1000
        mov bx, 1       ; 1 ciclo
        mov ax, 1       ; 1 ciclo, 0 se accoppiata
        endm
        ret
test1   endp
```

Questo test fornisce un risultato pari a circa 1006 cicli. Si può anche cercare di eliminare il sovraccarico dovuto alle istruzioni CALL e RET ma nella maggior parte dei casi questo non è necessario. In questo caso si vede che la coppia di istruzioni MOV richiede un solo ciclo e non due. Le macro di ripetizione sono un modo molto comodo per duplicare il codice e verificarne la velocità. Tuttavia, sul Pentium, occorre assicurarsi di capire esattamente cosa si sta effettivamente migliorando.

Utilizzando questo metodo vi sono alcuni fattori che possono provocare imprecisioni:

- Il codice e i dati del timer modificano il contenuto delle due cache (codice e dati).
- Questo può rallentare leggermente la procedura. Non si deve cercare di valutare le prestazioni di codice quando questo si avvicina alle dimensioni massime della cache del codice (8 KB) poiché si otterranno risultati non significativi.
- I risultati sono influenzati da interrupt, tempi di caricamento nella memoria cache e dalle dimensioni e dal tipo della cache secondaria.
- Il caricamento del programma nella cache prima dell'esecuzione fa sembrare il codice più veloce di quanto sia in realtà.

Lo scopo di questo metodo di valutazione è quello di mostrare le massime prestazioni di un blocco di codice quando le istruzioni vengono accoppiate in modo ottimale. Questo aiuta a determinare se le istruzioni sono disposte in modo corretto tenendo anche conto dei vari ritardi.

A seconda dello scopo per il quale si esegue il test, si possono utilizzare dati diversi ogni volta che viene richiamata una procedura:

```
lea si, string1
lea di, string1a
        mov     cx, 10000
        call    test2           ; precaricamento della cache con il codice e i dati
        lea     si, string2
        lea     di, string2a
        mov     cx, 10000
        call    timer_on        ; avvio del timer
        call    test2           ; esecuzione del test
        call    timer_off       ; arresto del timer
        ...
test2   proc
        mov     al, [si]        ; test di una copia di stringa
        inc     si
        mov     [di], al
        inc     di
```

```
        test    al, al
        loopne  test2
        ret
test2   endp
```

In questo caso, con una stringa estesa e un piccolo ciclo, non è necessario richiamare la procedura di test prima di verificarne le prestazioni poiché la differenza nel numero di istruzioni accoppiate risulta irrilevante rispetto al numero di dichiarazioni del ciclo.

A seconda di ciò che si intende misurare, vi sono alcune regole generali che assicurano che venga misurata l'effettiva velocità di esecuzione del programma:

- il codice e i dati devono essere allineati alle dword;
- le stringhe devono essere sufficientemente piccole per rientrare interamente nella cache dei dati;
- tutto il codice e i dati dovrebbero essere precaricati nella memoria cache;
- si deve utilizzare codice che rappresenti l'applicazione (ad esempio se si spostano dati fra segmenti occorre verificare se durante l'esecuzione il programma avrà a che fare con dati disallineati).

Le funzioni del timer software

Quello che segue è un elenco di tutte le funzioni della libreria fornita sul disco. In particolare vengono fornite sei versioni della libreria:

TTIMER.lib	Modello Tiny
STIMER.lib	Modello Small
CTIMER.lib	Modello Compact
MTIMER.lib	Modello Medium
LTIMER.lib	Modello Large
TIMER32.lib	Modello per la modalità protetta a 32 bit

Ad esempio, per utilizzare il timer in un programma DOS per il modello di memoria small si deve eseguire il link con il seguente comando:

```
C:> link esempio,,,stimer;
```

Se invece si deve utilizzare il timer per un programma a 32 bit operante in modalità protetta (vedere il Capitolo 18) si dovrà eseguire il linker con il seguente comando:

```
C:> link esempio,,,timer32;
```

timer_init

La funzione timer_init inizializza le variabili di sistema per eseguire test multipli sullo stesso codice. Consultare timer_show_average. Questa funzione non è necessaria nel caso di una semplice verifica di una sezione di codice e di stampa immediata dei risultati.

timer_on
Avvia il timer. In modalità reale, prima di richiamare questa funzione dovrebbero essere disabilitati gli interrupt (utilizzando CLI). In modalità protetta CLI non può essere utilizzata poiché provoca un errore di protezione generale.

timer_off
Ferma il timer che memorizza il tempo trascorso e che potrà in seguito essere utilizzato a piacere. Questa funzione restituisce nel registro AL uno dei seguenti codici:
0 successo;
1 errore, overflow del timer (evento troppo lungo, la lunghezza massima è di circa 0.0549 secondi);
2 errore, underflow del timer (evento troppo breve, la lunghezza minima è di circa 0.0000025 secondi ovvero 2.5 microsecondi);
3 errore, sovraccarico troppo esteso (è simile all'underflow: il sovraccarico è dovuto alla routine interna che è molto più lunga dell'evento misurato; probabilmente l'evento dura meno di 2.5 microsecondi).

timer_show
Visualizza i risultati dell'ultimo evento verificato. I risultati vengono visualizzati nel formato standard, ovvero in microsecondi. Tale formato può essere modificato con la funzione timer_set_format.

timer_show_ticks
Visualizza il risultato dell'ultimo evento verificato. I risultati vengono visualizzati in numero di cicli di clock. Ogni ciclo di clock dura approssimativamente 0.8381 microsecondi.

timer_show_microseconds
Visualizza il risultato dell'ultimo evento verificato. I risultati vengono visualizzati in microsecondi.

timer_show_average
Visualizza la media dell'ultimo gruppo di eventi verificati. I risultati vengono visualizzati nel formato standard ovvero in microsecondi. Tale impostazione può essere modificata utilizzando la funzione timer_set_format. Vedere timer_init.

timer_set_format
Imposta il formato di output delle funzioni timer_show e timer_show_average; AL=0 per i cicli di clock e AL=1 per il formato in microsecondi.

timer_write
Converte un valore da cicli di clock a una stringa ASCII e scrive la stringa risultante sul file specificato o sullo schermo. Gli input sono:

AX cicli di clock
BX handle del file per l'ouput (1 = stdout)

timer_ticks_to_ascii
Converte il valore specificato in cicli di clock in una stringa ASCII. Gli input sono:

AX cicli di clock
BX puntatore a un buffer di 5 byte per una stringa ASCII

timer_ticks_to_microsec
Converte i cicli di clock in microsecondi. L'input è:

AX cicli di clock

Mentre l'output è:

AX microsecondi

12.4 Variazioni percentuali della velocità

È possibile dimostrare che le variazioni percentuali nella velocità si basano sulla seguente formula:

variazione percentuale = ((T1 - T2) / T1 - 1) x 100 (Formula 1)

dove: T1 = prima volta, T2 = seconda volta.

I valori possono essere specificati in qualsiasi unità (ovvero in cicli oppure in secondi o microsecondi). Ad esempio, un cambiamento da 20 secondi a 10 secondi corrisponde a un miglioramento del 100%. D'altra parte se si preferisce vedere le cose da un altro punto di vista, ovvero considerare il fatto che è stata eliminata solo la metà del tempo di esecuzione, il miglioramento dovrebbe essere del 50%; dunque si può utilizzare anche la seguente formula:

variazione percentuale = ((T1 - T2) / T1) x 100 (Formula 2)

Tuttavia, quando i confronti successivi vengono eseguiti su codice che richiede meno cicli, questi miglioramenti sembreranno essere inferiori. Ad esempio si può consultare la seguente tabella.

	T1	T2	Formula 1	Formula 2	Formula 2 (T1 = precedente T2)
1.	20	10	100%	50%	
2.	20	8	150%	60%	20%
3.	20	6	233%	70%	25%
4.	20	4	400%	80%	33%
5.	20	2	900%	90%	50%

Con la seconda formula, utilizzando come base il valore T2 precedente, viene mostrato l'aumento di importanza di ogni secondo eliminato a mano a mano che il tempo diminuisce. Tuttavia, parlando di miglioramento delle prestazioni di una porzione di codice rispetto a un'altra, è chiaro che 2 secondi sono un risultato 10 volte migliore rispetto a 20 secondi con un miglioramento del 900%.

Parte quarta

PROGRAMMAZIONE SUPERSCALARE PER IL PENTIUM

Capitolo 13

Informazioni di base sull'ottimizzazione

Sulla base di quanto detto nei capitoli precedenti, è giunto il momento di vedere qualche esempio di ottimizzazione.

Il processo di "ottimizzazione" parte molto presto nel ciclo di sviluppo di un programma. In realtà dovrebbe partire prima della scelta dell'algoritmo. Teoricamente i requisiti del progetto dovrebbero dire chiaramente i criteri prestazionali da seguire ma ciò accade raramente. Il progetto include, fra gli altri dettagli, la scelta delle strutture dei dati e degli algoritmi. Le strutture dei dati e gli algoritmi avranno effetti notevoli sulle prestazioni, almeno nella maggior parte dei casi.

Dunque è fondamentale scegliere l'algoritmo corretto. Nella maggior parte dei casi l'ottimizzazione prevede la sostituzione di buone istruzioni con ottime istruzioni.

In questo capitolo e nel successivo si cercherà di mostrare varie tecniche di ottimizzazione del codice che manipola le stringhe. Ma anche se molte routine operano su stringhe ASCII, i concetti esposti hanno un'importanza molto più generale. In molti casi bastano poche modifiche per applicare le stesse tecniche a numeri o dati grafici.

Come si noterà l'interesse principale è legato all'ottimizzazione dei cicli più piccoli. Questa scelta ha varie ragioni. Innanzi tutto i cicli di codice più piccoli sono ideali per apprendere le tecniche di programmazione superscalare. In secondo luogo tali cicli sono molto utilizzati in programmazione. Infine l'ottimizzazione dei cicli più interni di una routine fornisce in genere il miglior livello di ottimizzazione.

Se si ottimizza codice che non verrà mai eseguito in un ciclo si corre il rischio di non velocizzare affatto il codice.

13.1 Ottimizzazione delle istruzioni che operano sulle stringhe

Si consideri la seguente porzione di codice che può essere stata realizzata nei primi anni '80 per l'8086. Questo codice copia una stringa ASCIIZ (una stringa di caratteri ASCII terminata dal carattere nullo).

```
lbl:                                          (Listato 13.1)
        lodsb           ; carica un byte
        stosb           ; salva un byte
        or     al, al   ; verifica che sia diverso da null
        jne    lbl      ; in caso negativo salta
```

Il alternativa si possono utilizzare le istruzioni MOV e INC corrispondenti alle istruzioni LODSB e STOSB (vedere Figura 13.1b). In questo modo non viene eseguita esattamente l'operazione originale, poiché STOSB utilizza normalmente il segmento ES. Aggiungendo l'uscita dal segmento alla seconda istruzione MOV presentata nella Figura 13.1b, si ottiene lo stesso effetto ma, su alcune CPU si allunga l'esecuzione di uno o due cicli. In tutti gli esempi, il codice è normalmente disposto in modo da eliminare le uscite dai segmenti o si suppone che il codice di inizializzazione abbia reso inutile tale operazione. Quando si rende necessario eseguire operazioni che operano su più segmenti, ad esempio una copia da un segmento a un altro, il codice risulterà più lento. Questo argomento verrà però presentato alla fine di questa parte del manuale.

A mano a mano che sono state introdotte nuove CPU, questo codice è stato riesaminato per vedere se era conveniente modificarlo. Dall'8088 al 386 le istruzioni sulle stringhe sono in genere migliori o uguali alle semplici istruzioni di caricamento e salvataggio. Sul 486 (che ha alcune caratteristiche delle macchine RISC), in genere le istruzioni di caricamento e salvataggio, essendo più semplici, tendono a comportarsi meglio. Anche se le operazioni sulle stringhe sul 486 non sono così veloci, rimangono comunque più compatte (in questo caso 6 byte invece di 11). Nel caso del Pentium l'aumento di velocità è notevolissimo (da 6 a 3 cicli, un aumento teorico pari al 100%) mentre nel caso del 486 l'aumento di prestazioni è pari al 75% (da 14 a 8 cicli). In queste cifre si è indicato il numero di cicli per ogni istruzione sul Pentium assumendo che non avvenga alcun accoppiamento di istruzioni. Nella colonna relativa all'esecuzione con accoppiamento, si assume che le istruzioni vengano accoppiate secondo le relative regole. Dunque per un'istruzione eseguita nella pipe V verrà mostrato solo il numero di cicli aggiuntivi richiesti oltre a quelli richiesti dall'istruzione in esecuzione nella pipe U (normalmente questo valore è 0).

Come si è visto, alcune istruzioni per le stringhe sono più lente rispetto alle più semplici istruzioni di spostamento e incremento quanto queste ultime possono essere accoppiate e dunque eseguite in un unico ciclo. Inoltre, la combinazione CMP/Jcc (o TEST/Jcc) può essere accoppiata in modo da essere eseguita in un unico ciclo. A tale proposito si consulti la Tabella 13.1 (le istruzioni di ripetizione relative alle stringhe verranno discusse in seguito).

Come esercizio si può provare a ottimizzare ulteriormente il codice riportato nella Figura 13.1 di passando da 3 cicli per byte a 2 cicli per byte.

Il prossimo esempio (Figura 13.2) copia una stringa ASCIIZ con un limite legato alla lunghezza massima della stringa. Questo esempio mostra che LOOPNE (e LOOPE) è molto più lenta rispetto alle equivalenti istruzioni Jcc/DEC/Jcc. Ancora una volta le istruzioni LODSB e STOSB sono state sostituite da istruzioni MOV/INC. Questo riduce il numero di cicli sul Pentium da 13 a 4 con un aumento di velocità del 225%.

(a) (Listato 13.2)

			8088	286	386	486	Pent.	accoppiate	byte
loopl:									
	lodsb		16	5	5	5	2	2	1
	stosb		15	3	4	5	3	3	1
	or	al, al	3	2	2	1	1	1	2
	jne	loop1	16	8	8	3	1	1	2
			50	18	19	14	7	7	6

(b) (Listato 13.3)

			8088	286	386	486	Pent.	accoppiate	byte
loop2:									
	mov	al, [si]	17	5	4	1	1	1	2
	inc	Si	3	2	2	1	1	0	1
	mov	[di], al	18	4	2	1	1	1	2
	inc	di	3	2	2	1	1	0	1
	cmp	al, 0	4	3	2	1	1	1	2
	jne	loop2	16	9	8	3	1	0	2
			61	25	20	8	6	3	11

Figura 13.1 Copia di una stringa ASCIIZ con il limite della massima lunghezza della stringa.

Tabella 13.1 Cicli di CPU per le istruzioni di manipolazione delle stringhe.

	486	**Pentium**	**Pairing**
MOV reg, mem	1	1	UV
MOV mem, reg	1	1	UV
INC/DEC reg	1	1	UV
TEST/CMP reg, reg/imm	1	1	UV
Jcc	1/3*	1**	PV
LODS	5	2	NP
STOS	5	3	NP
REP MOVS	3	1	NP
REP STOS	4	1	NP
REPE/NE CNPS	7	4	NP
REPE/NE SCAS	5	4	NP

Nel prossimo esempio si proverà a copiare da una locazione a un'altra una stringa di lunghezza nota. Nei primi esempi è già stato presentato il numero di cicli di CPU prodotti da ciascuna istruzione e dunque tali valori non verranno ripresentati per i prossimi esempi. Verranno visualizzati solo i cicli necessari sul Pentium, in quanto questi forniscono le informazioni più utili. Ecco un ottimo modo per copiare una stringa di lunghezza fissa:

```
rep     movsb                    ; 1 ciclo per byte
```

Ecco come è possibile eseguire la stessa operazione utilizzando solo istruzioni semplici:

(Listato 13.5)

```
lbl:                        ; cicli
        mov    al, [si]     ; 1   legge un byte
        inc    si           ; 0   fa avanzare il ptr
        mov    [di], al     ; 1   memorizza un byte
        inc    di           ; 0   fa avanzare il ptr
        dec    cx           ; 1   decrementa il contatore del ciclo
        jnz    lbl          ; 0   finché CX non è = 0
                            ; —
                            ; 3 cicli per byte
```

Questo esempio mostra che non tutte le operazioni complesse sulle stringhe possono sfruttare le pipeline del Pentium risultando quindi più veloci. D'altra parte, forse l'istruzione REP MOVSB utilizza entrambe le pipe del Pentium (Intel non documenta l'implementazione delle istruzioni ma dice che il Pentium utilizza entrambe le pipe

(a) (Listato 13.3)

			8088	286	386	486	Pent.	accopp.
loop3:								
	lodsb		16	5	5	5	2	2
	stosb		15	3	4	5	3	3
	or	al, al	3	2	2	1	1	1
	loopne	loop3	19	10	13	9	7	7
			53	20	24	20	13	13

(b) (Listato 13.4)

			8088	286	386	486	Pent.	accopp
loop4:								
	mov	al, [si]	17	12	4	1	1	1
	inc	si	3	2	2	1	1	0
	mov	[di], al	18	9	2	1	1	1
	inc	di	3	2	2	1	1	0
	cmp	al, 0	4	3	2	1	1	1
	je	exit4	4	3	3	1	1	0
	dec	cx	3	2	2	1	1	1
	jnz	loop4	16	9	8	3	1	0

Figura 13.2 Copia di una stringa ASCIIZ con una stringa di lunghezza massima.

anche per le istruzioni non accoppiabili). Ma vi sono metodi anche più veloci per copiare una stringa. Se fosse noto che il numero di byte è pari, si potrebbe utilizzare solo la metà del numero di word.

```
shr     cx, 1               ; divide il numero di byte per 2 (resto in CF)     (Listato 13.6)
        rep     movsw       ; sposta le word
        rcl     cx, 1       ; ripristina CF in CX
        rep     movsb       ; sposta 0 o 1 byte
```

oppure

```
shr     cx, 1               ; divide il numero di byte per 2 (resto in CF)
        rep     movsw       ; sposta le word
        jnc     exit        ; verifica CF, esce se è =0
        movsb               ; sposta 1 byte
```

Si possono trasferire anche quattro byte per volta:

```
mov     ax, cx              ; salva una copia del contatore dei byte            (Listato 13.7)
shr     cx, 2               ; divide per 4 per ottenere il numero di dword
rep     movsd               ; sposta le dword
mov     cx, ax              ; ripristina il numero di byte
and     cx, 3               ; prende i 2 bit inferiori (resto della div per 4)
rep     movsb               ; sposta da 0 a 3 byte
```

Il codice della Figura 13.1b può essere riscritto in modo da gestire word o dword invece di semplici byte. Ogni metodo ha i suoi vantaggi e i suoi svantaggi. Ad esempio si potrebbe riscrivere il codice nel seguente modo:

```
mov     ax, [si]        ; 1     legge una word                              (Listato 13.8)
add     si, 2           ; 0     fa avanzare il ptr
mov     [di], ax        ; 1     memorizza una word
add     di, 2           ; 0     fa avanzare il ptr
cmp     al, 0           ; 1     verifica del null nel 1o byte
je      exit            ; 0     esce se è la fine della stringa
cmp     ah, 0           ; 1     verifica del null nel 2o byte
jnz     lbl             ; 0     continua se non è a fine stringa
```

Questo codice copia un byte ogni due cicli invece che ogni tre cicli come nel codice precedente. Lo svantaggio è che vi è una probabilità del 50% che questo ciclo copi un byte aggiuntivo oltre il carattere nullo. Questo potrebbe in alcuni casi presentare dei problemi. L'esempio seguente non presenta tale effetto collaterale:

```
mov     ax, [si]        ; 1     legge una word                              (Listato 13.9)
add     si, 2           ; 0     fa avanzare il ptr
mov     [di], al        ; 1     memorizza il 1o byte
add     di, 2           ; 0     fa avanzare il ptr
cmp     al, 0           ; 1     verifica del null nel 1o byte
je      exit            ; 0     esce se è a fine stringa
mov     [di-1], ah      ; 1     memorizza il 2o byte
cmp     ah, 0           ; 0     verifica del null nel 2o byte
jne     lbl             ; 1     continua se non è a fine stringa
exit:                   ; —
                        ; 5 cicli — 2.5 cicli per byte
```

Lo svantaggio dovuto alla necessità di assicurare che non venisse copiato un byte aggiuntivo oltre il carattere nullo si può misurare in 0.5 cicli aggiuntivi per byte (2.5 cicli invece di 2.0). Tuttavia ecco un metodo che consente di copiare la stringa senza alcun byte aggiuntivo:

(Listato 13.10)

```
lbl                           ; cicli
        mov   ax, [si]        ; 1     legge una word
        add   si, 2           ; 0     fa avanzare il puntatore
        cmp   al, 0           ; 1     verifica del null nel primo byte
        je    exit2           ; 0     esce se è la fine della stringa
        mov   [di], ax        ; 1     salva 2 byte
        add   di, 2           ; 0     fa avanzare il puntatore
        cmp   ah, 0           ; 1     verifica del null nel secondo byte
        jne   lbl             ; 0     continua se non è a fine stringa
exit:                         ; —
        jmp   exit3           ; 4 cicli — 2 cicli per byte
exit2:
        mov   [di], al
exit3:
```

Sul Pentium è particolarmente utile ricercare questo tipo di ottimizzazione. Un risparmio di uno o due cicli di CPU per ogni ciclo di istruzioni su un 8088 e perfino su un 386 potrebbe non aumentare in modo sensibile le prestazioni del blocco di codice; se invece si riesce a risparmiare uno o due cicli di CPU su un Pentium si ottiene un grande vantaggio quando il ciclo di istruzioni è composto da 3-10 cicli di CPU.

Ecco il codice precedente contenente le istruzioni che consentono di copiare i dati da un segmento a un altro:

(Listato 13.11)

```
lbl:                          ; cicli
        mov   ax, [si]        ; 1     legge una word
        add   si, 2           ; 0     fa avanzare ptr
        cmp   al, 0           ; 1     verifica null nel 1o byte
        je    exit2           ; 0     esce se a fine stringa
        mov   ES:[di], ax     ; 2     memorizza due byte
        add   di, 2           ; 0     fa avanzare ptr
        cmp   ah, 0           ; 1     verifica null nel 2o byte
        jne   lbl             ; 0     continua se non è a fine stringa
exit:                         ; —
        jmp   exit3           ; 5 cicli — 2.5 cicli per byte
exit2:
        mov   ES:[di], al
exit3:
```

Poiché l'istruzione di uscita dal segmento (ES:) deve essere eseguita nella pipe U, vi è una penalità di un ciclo. Se l'uscita dal segmento fosse stata aggiunta a un'istruzione da eseguire nella pipe V, non ne sarebbe stato possibile l'accoppiamento e questo avrebbe provocato una penalizzazione di due cicli.

Quando si presenta un esempio di codice a 32 bit, si intende che debba essere utilizzato in un segmento di codice a 32 bit (USE32). Generalmente questo codice deve essere eseguito da Windows NT, OS/2 2.x o da un extender DOS a 32 bit (per un

esempio di codice DPMI a 32 bit si consulti il Capitolo 18). Anche se è possibile utilizzare insieme istruzioni a 32 bit e istruzioni a 16 bit, solo le istruzioni nel formato nativo operano a piena velocità. Le istruzioni non native provocano un ritardo di un ciclo e possono essere accoppiate solo nella pipe U. Anche se esiste questa restrizione, in alcuni casi può essere vantaggioso utilizzare insieme codice a 16 e a 32 bit.

Ecco un esempio di copia di una stringa realizzato con codice a 32 bit:

(Listato 13.12)

```
                            ; cicli
lbl:                        ; a 32 bit
        mov     eax, [esi]  ; 1
        add     esi, 4      ; 0
        mov     [edi], eax  ; 1
        add     edi, 4      ; 0
        cmp     al, 0       ; 1
        jz      exit        ; 0
        cmp     ah, 0       ; 1
        jz      exit        ; 0
        bswap   eax         ; 1
        cmp     al, 0       ; 1
        jz      exit        ; 0
        cmp     ah, 0       ; 1
        jne     lbl         ; 0
exit:                       ; —
                            ; 7 cicli — 1.75 cicli per byte
```

Questo codice scrive fino a tre byte aggiuntivi e richiede solo 1.75 cicli per byte. In pratica si approssima alla velocità dell'istruzione REP MOVSB. Ma occorre notare che questo esempio di copia di una stringa ricerca la fine della stringa in ogni byte. Come è possibile accelerarlo?

L'unico metodo in cui è possibile accelerare questa operazione consiste nell'aggiungere altri caratteri nulli alla fine della stringa. Ad esempio, per gestire due byte per volta sarebbe necessario aggiungere 2 byte nulli aggiuntivi mentre nel caso di copie a quattro byte, sarebbe necessario aggiungere 6 byte nulli aggiuntivi. In questo caso probabilmente il codice più veloce sarebbe:

(Listato 13.13)

```
                            ; cicli
lbl:                        ; a 32 bit
        mov     eax, [esi]  ; 1
        add     esi, 4      ; 0
        mov     [edi], eax  ; 1
        add     edi, 4      ; 0
        test    eax, eax    ; 1
        jne     lbl         ; 0
exit:                       ; —
                            ; 3 cicli — 0.75 cicli per byte
```

Questi esempi mostrano che quando è possibile controllare le specifiche dei dati si può più che raddoppiare la velocità del codice. Alcuni possono ritenere che i problemi e le soluzioni presentate finora siano state disposte in un ordine errato. La scelta è intenzionale poiché si intende incoraggiare a provare varie soluzioni ai problemi, a

trasformarle in codice per poi eseguirne la verifica. Come si è visto, anche per la semplice operazione di copia di una stringa non vi è una singola soluzione e lo stesso avviene con i problemi più complessi.

Questi esempi di copia di stringhe non sono che un assaggio delle possibilità di ottimizzazione del Pentium. Inoltre si ricordi che queste operazioni non si limitano alle stringhe di testo. In questo capitolo si è presa un'operazione apparentemente semplice, utilizzabile un qualsiasi applicazione e si è mostrato come è possibile modificarla per produrre un notevole incremento prestazionale, pari al 900%. Si è partiti con un ciclo di istruzioni da 50 cicli di CPU sull'8088 (7 sul Pentium) e si è ottenuto un miglioramento fino a 1.75 cicli di CPU e a 0.75 cicli di CPU introducendo una lieve modifica nel formato dei dati.

Capitolo 14

Ricerca e traduzione di stringhe

Spesso si ritiene di conoscere approfonditamente il set di istruzioni dei microprocessori 80x86 quando in realtà ci si è trovati a utilizzare solo alcune delle istruzioni disponibili. Se non si prova a servirsene e a scontrarsi con gli errori che provocano, le descrizioni potrebbero sembrare mappe stradali prive della metà delle strade.

14.1 Ricerca di una stringa

Questo capitolo parte da una descrizione delle ottimizzazioni possibili nella ricerca di stringhe.

Si è lavorato molto per rendere il più possibile efficienti gli algoritmi di ricerca di stringhe. Il problema consiste in genere nel ricercare una determinata sequenza di caratteri in una stringa di testo (se la ricerca avviene in un grosso file, questo verrà letto blocco per blocco in un buffer). Le tecniche più complesse come l'algoritmo di Boyet-Moore e l'algoritmo di Knuth-Morris-Pratt si dimostrano teoricamente molto efficienti ma il metodo della "forza bruta" è stato ampiamente ottimizzato per l'architettura 80x86. Questo metodo prevede la scansione della stringa di testo alla ricerca del primo carattere che compone la sequenza ricercata. Quando viene trovato tale carattere, la verifica procede con il carattere successivo. Anche la scansione quindi procede con il carattere seguente della stringa. Questo algoritmo funziona bene quando la scansione trova un basso numero di sequenze errate e quando la sequenza ricercata è piuttosto piccola. La velocità è dovuta al fatto che il ciclo interno di questo tipo di routine si riduce a una sola istruzione:

```
                ;        8088   286   386   486   Pentium
repne scasb     ; cicli: 15     8     8     5     4
```

Come si può vedere, l'istruzione REPxx SCASB è sempre stata piuttosto veloce. Tuttavia il Pentium modifica leggermente la situazione. La logica di previsione dei salti e le possibilità di accoppiamento delle istruzioni del Pentium consentono di sostituire all'istruzione REPxx SCASB il seguente codice che risulta un ciclo più veloce:

(Listato 14.1)

```
                  ; (nota: DL contiene il byte da confrontare)
lbl:              ; cicli
      mov  al, [di] ; 1  legge un carattere
      inc  di       ; 0  fa avanzare il puntatore
      cmp  al, dl   ; 1  confronta con il carattere
      je   exit     ; 0  esce se sono uguali
      dec  cx       ; 1
      jnz  lbl      ; 0
                    ; —
                    ; 3 cicli per byte
```

Questo ciclo di istruzioni richiede tre cicli di CPU per byte ma il processo di ottimizzazione è appena iniziato. Vi sono sei punti di esecuzione (due per ciclo) e due di essi sono dedicati al controllo del ciclo stesso. Si può rendere questa porzione di codice più veloce elaborando due byte per ciclo e aggiungendo altre porzioni di codice per la gestione di un ulteriore byte:

(Listato 14.2)

```
lbl:                ; cicli
      mov  ax, [di] ; 1 legge due caratteri
      add  di, 2    ; 0 fa avanzare il puntatore
      cmp  al, dl   ; 1 confronta il 1o con il carattere in scansione
      je   exit1    ; 0 esce (non è illustrata l'etichetta) se trova una corrispondenza
      cmp  ah, dl   ; 1 confronta il 1o con il carattere in scansione
      je   exit2    ; 0 esce (non è illustrata l'etichetta) se trova una corrispondenza
      dec  cx       ; 1
      jnz  lbl      ; 0
                    ; —
                    ; 4 cicli, 2 cicli per byte
```

Continuando in questo processo, è possibile elaborare quattro byte per ciclo. Questo crea una maggiore quantità di codice ausiliario per la gestione dei byte da 0 a 3 ma consente di risparmiare solo 0,25 cicli per byte (circa il 12%):

(Listato 14.3)

```
lbl:                    ; cicli
      mov  ax, [di]     ; 1 legge due caratteri
      mov  bx, [di+2]   ; 0 legge altri due caratteri
      cmp  al, dl       ; 1 confronta il 1o con il carattere
      je   exit1        ; 0 esce se sono uguali
      cmp  ah, dl       ; 1 confronta il 2o con il carattere
      je   exit2        ; 0 esce se sono uguali
      cmp  bl, dl       ; 1 confronta il 3o con il carattere
      je   exit3        ; 0 esce se sono uguali
      cmp  bh, dl       ; 1 confronta il 4o con il carattere
      je   exit4        ; 0 esce se sono uguali
      add  di, 4        ; 1 fa avanzare il puntatore
      dec  cx           ; 0
```

```
        jnz     lbl             ; 1
                                ; 7 cicli, 1.75 cicli per byte
```

Nella maggior parte dei casi, il metodo della forza bruta che utilizza REPNE SCASB fornisce prestazioni eccellenti sui chip 8088-486. Sul Pentium il metodo della forza bruta è invece inferiore ai risultati ottenibili utilizzando istruzioni più semplici. Quest'ultima implementazione ha infatti una velocità più che doppia rispetto all'istruzione REPNE SCASB.

Nel codice precedente vi è però un problema. Come si è detto nel Capitolo 10, un conflitto nei banchi di memoria cache dei dati può provocare un ritardo di un ciclo. L'unico modo per evitare questo nel codice precedente consiste nell'assicurare che l'indirizzo iniziale di tale codice si trovi sui limiti di una word ma non di una dword. Questo avviene naturalmente nel 25% dei casi. Il problema può essere risolto con il seguente codice:

```
lbl:                            ; cicli                          (Listato 14.4)
        mov     ax, [di]        ; 1 legge due caratteri
        add     di, 4           ; 0 fa avanzare puntatore
        cmp     al, dl          ; 1 confronta il 1o con il carattere
        je      exit1           ; 0 esce se sono uguali
        mov     bx, [di-2]      ; 1 legge altri due caratteri
        cmp     ah, dl          ; 0 confronta il 2o con il carattere
        je      exit2           ; 1 esce se sono uguali
        cmp     bl, dl          ; 1 confronta il 3o con il carattere
        je      exit3           ; 0 esce se sono uguali
        cmp     bh, dl          ; 1 confronta il 4o con il carattere
        je      exit4           ; 0 esce se sono uguali
        dec     cx              ; 1
        jnz     lbl             ; 0
                                ; 7 cicli, 1.75 cicli per byte
```

A questo metodo possono essere applicate varie ottimizzazioni. Una delle più comuni, quando si esegue una ricerca all'interno di un file di testo ASCII, consiste nel ricercare un carattere diverso dal primo carattere della sequenza ricercata. Il proposito è quello di ricercare un carattere che appaia il minor numero di volte possibile nella stringa. Ad esempio, se si cerca la parola "acqua" il risultato ottimale si ottiene ricercando la lettera "q" poiché è probabile che appaia con una frequenza inferiore rispetto alle altre lettere che compongono la parola. Naturalmente, prima di applicare questa ottimizzazione sarebbe consigliabile conoscere la frequenza di distribuzione dei dati sui quali viene eseguita la ricerca.

14.2 Traduzione di stringhe

In questo esempio si partirà da un ciclo che converte una stringa in lettere minuscole. Lo stesso ciclo può essere utilizzato per eseguire qualsiasi tipo di traduzione modificando solo la tabella di traduzione utilizzata.

```
        lea si, string          ; carica il puntatore alla stringa          (Listato 14.5)
        mov   cx, max_str_len   ; legge la lunghezza max della stringa
        lea   bx, tbl           ; carica il puntatore alla tabella di traduzione
                                ; tbl è una tabella di traduzione composta da 256 byte
                                ; utilizzabile per qualsiasi scopo, ad esempio
                                ; la conversione in lettere minuscole.
lbl:                            ; cicli
        lodsb                   ; 2 legge un byte
        xlatb                   ; 3 traduce in tabella
        mov   [si-1], al        ; 1 memorizza la traduzione
        test al, al             ; 1 test di fine stringa
        loopnz lbl              ; 7 continua il ciclo
                                ; —
                                ; 14 cicli
```

Questo ciclo è molto semplice: legge una stringa di caratteri un carattere alla volta, produce i caratteri utilizzando una tabella, salva il nuovo valore e verifica che non sia stata raggiunta la fine della stringa (un carattere nullo). Inoltre, in CX viene conservato il numero massimo di caratteri che può comporre la stringa e che quindi pone fine al ciclo. Se fosse nota la lunghezza della stringa, si potrebbe scrivere il codice nel modo seguente:

```
lbl:                        ; cicli                          (Listato 14.6)
        lodsb               ; 2 legge un byte
        xlatb               ; 3 lo traduce in tabella
        mov   [si-1], al    ; 1 salva la traduzione
        loop lbl            ; 5 continua il ciclo
                            ; —
                            ; 11
```

Per ottimizzare questo codice per il Pentium, la prima cosa da fare è quella di convertire il tutto in istruzioni semplici. Ecco due modi leggermente diversi per eseguire l'operazione:

```
lbl:                        ; cicli                          (Listato 14.7)
        mov   bl, [si]      ; 1
        mov   al, tbl[bx]   ; 2 <— blocco AGI
        mov   [si], al      ; 1 <— conflitto di registro, non accoppiabile
        inc   si            ; 0
        test al, al         ; 1
        jz    done          ; 0
        dec   cx            ; 1
        jnz   lbl           ; 0
done:                       ; —
                            ; 6
```

```
lbl:                        ; cicli                          (Listato 14.8)
        mov   bl, [si]      ; 1
        inc   si            ; 0
        mov   al, tbl[bx]   ; 2 <— blocco AGI
        mov   [si-1], al    ; 1 <— conflitto di registro, non accoppiabile
        test al, al         ; 0
```

```
        jz      done        ; 1
        dec     cx          ; 0
        jnz     lbl         ; 1
done:                       ; —
                            ; 6
```

Anche se entrambi tentano di produrre codice che risulti veloce più del doppio rispetto all'originale (6 cicli contro 4 cicli) entrambi i tentativi sono suscettibili di miglioramenti (eliminando il test di fine stringa è possibile risparmiare un ciclo ma deve essere nota la lunghezza della stringa). Entrambi i cicli di istruzioni presentano un blocco AGI e un problema nell'accoppiamento delle istruzioni a causa di un conflitto nei registri. Il conflitto nei registri e il blocco AGI possono essere eliminati caricando in anticipo il primo carattere e leggendo il secondo carattere già durante la prima esecuzione del ciclo, ovvero durante la manipolazione del primo carattere:

(Listato 14.9)

```
        mov     bl, [si]        ; legge il primo byte
        inc     si
lbl:                            ; cicli
        mov     al, tbl[bx]     ; 1 traduzione in tabella
        mov     bl, [si]        ; 0 legge il secondo byte
        mov     [si-1], al      ; 1 memorizza la traduzione del primo byte
        inc     si              ; 0 fa avanzare il puntatore
        test al, al             ; 1 verifica Null
        jz      done            ; 0 esce se è a fine stringa
        dec     cx              ; 1
        jnz     lbl             ; 0 continua il ciclo
done:                           ; —
                                ; 4
```

Questo elimina il problema e si può pensare di avere raggiunto la situazione ottimale: le istruzioni possono essere accoppiate e vengono utilizzate 8 istruzioni su 8. Ma si può fare di meglio.

14.3 **Programmazione atomica**

Conoscendo le massime prestazioni che è possibile raggiungere, si riesce a capire quando è concluso il lavoro di ottimizzazione. Ecco un metodo utilizzabile. La prima fase consiste nella trasformazione delle istruzioni complesse in istruzioni "atomiche". Le istruzioni atomiche sono singole funzioni della CPU che non possono essere ulteriormente semplificate (secondo i principi delle operazioni RISC). Se vengono scelte e disposte correttamente, le operazioni atomiche possono essere eseguite due alla volta, una per ogni pipeline. Per produrre una stringa sono richieste quattro fasi atomiche: lettura di un carattere;

- lettura della traduzione;
- memorizzazione del risultato;
- avanzamento del puntatore alla posizione successiva.

Se si dispongono correttamente queste quattro operazioni nelle due pipeline, queste potranno essere eseguite in due soli cicli di CPU. La gestione del ciclo di istruzioni risiede due operazioni atomiche, un decremento e un salto condizionale. La verifica della condizione di fine stringa (carattere nullo) richiede altre due operazioni atomiche. Dunque teoricamente è possibile scrivere un ciclo di traduzione di una stringa che richieda solo quattro cicli per byte. Occorre notare che questo risultato si è ottenuto immediatamente.

Ma non sempre la semplicità di questo metodo porta con sicurezza a risultati ottimali. In particolare non si deve assumere di aver sempre l'algoritmo migliore e di aver identificato correttamente le operazioni atomiche. In questo caso è possibile ottenere un risultato migliore rispetto a quattro cicli di CPU per byte poiché in realtà sono necessari solo tre cicli: le istruzioni di controllo del ciclo possono essere eliminate duplicando il codice o estendendo il ciclo delle istruzioni.

ATTENZIONE *Anche se l'espansione del ciclo è molto utile nei microprocessori dall'8088 al 486, sul Pentium si può correre il rischio di rallentare il codice.*

Le regole di accoppiamento delle istruzioni nella pipe U richiedono che l'istruzione abbia una lunghezza di un solo byte o che sia già stata eseguita dalla cache. Le istruzioni accoppiabili e lunghe un solo byte sono rare (vedere l'Appendice F). La prima volta che si esegue il ciclo, potrebbe non verificarsi alcun accoppiamento delle istruzioni con un raddoppio dei tempi di esecuzione. Questo tentativo di ottimizzazione estrema sarà dunque utile solo se applicato a cicli eseguiti molte volte.

Ecco un ciclo che elabora due byte per ogni dichiarazione:

(Listato 14.10)

```
        mov     bl, [si]
        inc     si
lbl:                            ; cicli
        mov     al, tbl[bx]     ; 1 traduce il primo byte
        mov     dx, [si]        ; 0 legge il secondo e il terzo byte
        mov     [si-1], al      ; 1 memorizza la traduzione del primo byte
        mov     bl, dl          ; 0 sposta il secondo byte
        test al, al             ; 1 verifica il Null
        jz      done            ; 0 esce alla fine della stringa
        mov     al, tbl[bx]     ; 1 traduce il secondo byte
        mov     bl, dh          ; 0 sposta il terzo byte per il ciclo successivo
        mov     [si], al        ; 1 memorizza la traduzione del secondo byte
        test al, al             ; 0 verifica il Null
        jz      done            ; 1 esce alla fine della stringa
        add     si, 2           ; 0 fa avanzare il puntatore
        dec     cx              ; 1
        jnz     lbl             ; 0 ripete il ciclo
done:                           ; —
                                ; 7 totale      3.5 per byte
```

Aggiungendo il codice di elaborazione di 4 byte per iterazione, si ottengono 13 cicli ovvero 3.25 cicli per byte. Occorre però tenere in considerazione che a mano a mano che aumenta la complessità di un ciclo (ovvero mano a mano che si aumenta il numero dei byte gestiti), aumenta anche la complessità del codice esterno al ciclo che deve gestire la situazione verificatasi a causa dei byte aggiuntivi letti. Talvolta è possi-

bile disporre le istruzioni in modo che i byte aggiuntivi possano essere gestiti saltando direttamente all'interno del ciclo:

(Listato 14.11)

```
        mov   bl, [si]          ; lettura anticipata del primo byte
        inc   si

        shr   cx, 1             ; divide il numero di byte per 2 per ottenere le word
        jnc   lbl               ; verifica che non vi sia un byte dispari
        inc   cx                ; gestisce il byte dispari incrementando il contatore,
        inc   si                ; sposta avanti il puntatore
        jmp   lbl2              ; quindi salta all'interno del ciclo

lbl:                            ; cicli
        mov   al, tbl[bx]       ; 1 traduce il primo byte
        mov   bl, [si]          ; 0 legge il secondo byte
        mov   [si-1], al        ; 1 memorizza la traduzione del primo byte
        add   si, 2             ; 0 fa avanzare il puntatore
        test al, al             ; 1 verifica del Null
        jz    done              ; 1 esce alla fine della stringa
lbl2:
        mov   al, tbl[bx]       ; 0 traduce il secondo byte
        mov   bl, [si-1]        ; 1 legge il primo byte per il prossimo ciclo
        mov   [si-2], al        ; 0 memorizza la traduzione del secondo byte
        test al, al             ; 1 verifica del Null
        jz    done              ; 0 esce alla fine della stringa
        dec   cx                ; 1
        jnz   lbl               ; 0 continua il ciclo
done:                           ; —
                                ; 7 totale      3.5 per byte
```

Per consentire l'ingresso all'interno del ciclo si è resa necessaria una manipolazione più complessa rispetto a una semplice ridisposizione delle istruzioni. Gli ultimi due esempi presentati avevano uno scopo. Si tratta dei primi tentativi per scrivere questa routine e mostrano alcune difficoltà e imperfezioni (non sempre il codice è perfetto). Dunque non si deve pensare che basti scrivere codice per la doppia pipeline per ottenere immediatamente il risultato ottimale. In determinate condizioni ci si potrebbe sentire scoraggiati e ricominciare da capo applicando altre regole e continuando a verificare il codice.

La prima traduzione di stringhe con due byte per ciclo è leggermente più convenzionale poiché fa avanzare il puntatore alla fine del ciclo di istruzioni. Inizialmente si può pensare che sia preferibile leggere contemporaneamente due byte. Ma iniziando a generare codice secondo questo principio risulta difficile trovare un metodo alternativo finché non si riesce a considerare le cose da un altro punto di vista. In particolare si deve considerare la possibilità di entrare direttamente all'interno del ciclo per gestire il byte aggiuntivo.

Eliminando i test di fine stringa si ottiene il seguente codice:

(Listato 14.12)

```
        mov   bl, [si]
        inc   si
lbl:                            ; cicli
        mov   al, tbl[bx]       ; 1 traduce il primo byte
```

```
        mov   bl, [si+6]      ; 0 legge il secondo byte (+7 rispetto al primo byte)
        mov   [si-1], al      ; 1 memorizza la traduzione del primo byte
        mov   al, tbl[bx]     ; 0 traduce il secondo byte
        mov   bl, [si+1]      ; 1 legge il primo byte per il prossimo ciclo
        mov   [si+6], al      ; 0 memorizza la traduzione del secondo byte
        add   si, 2           ; 1 fa avanzare il puntatore
        dec   cx              ; 0
        jnz   lbl             ; 1
                              ; —
                              ; 5 totale 2.5 per byte
```

Si consiglia di studiare in modo approfondito questo ciclo. Si tratta di un ottimo modello per istruzioni intervallate che suddividono una stringa in due flussi di dati distinti. Il problema è che la quinta e la sesta istruzione accedono entrambe alla memoria leggendo byte adiacenti. Questo può provocare un ritardo di un ciclo a causa di un conflitto nei banchi della memoria cache che contiene i dati. Il ritardo si verificherà il 100% delle volte se i dati sono allineati ai margini di una word e il 50% delle volte se i byte se i dati sono allineati su un byte dispari.

Un modo per correggere questa situazione consiste nel fare in modo che il secondo byte elaborato nel ciclo si trovi a cinque byte rispetto al primo. Ma questo accorgimento non elimina completamente i ritardi, poiché il conflitto nei banchi di memoria si verifica quando si deve leggere il primo byte nel ciclo successivo (che si trova a due byte di distanza rispetto al primo byte letto dal ciclo corrente). Se invece si partisse a 7 byte di distanza non vi sarebbe più alcun conflitto. Ma questo richiede l'uso dei codice particolare di gestione anticipata dei byte 2, 4 e 6:

(Listato 14.13)

```
        mov   bl, [si]
        inc   si
lbl:                          ; cicli
        mov   al, tbl[bx]     ; 1 traduce il primo byte
        mov   bl, [si+6]      ; 0 legge il secondo byte (+7 rispetto al primo byte)
        mov   [si-1], al      ; 1 memorizza la traduzione del primo byte
        mov   al, tbl[bx]     ; 0 traduce il secondo byte
        mov   bl, [si+1]      ; 1 legge il primo byte per il prossimo ciclo
        mov   [si+6], al      ; 0 memorizza la traduzione del secondo byte
        add   si, 2           ; 1 fa avanzare il puntatore
        dec   cx              ; 0
        jnz   lbl             ; 1
                              ; —
                              ; 5 totale 2.5 per byte
```

Oltre all'esempio precedente, l'autore ha tentato altri sei modi per modificare questo codice, fino a scoprire che l'accelerazione poteva essere ottenuta aggiungendo un'istruzione NOP:

(Listato 14.14)

```
lbl:                          ; cicli
        mov   al, tbl[bx]     ; 1 traduce il primo byte
        mov   bl, [si]        ; 0 legge il secondo byte
        mov   [si-1], al      ; 1 memorizza la traduzione del primo byte
        mov   al, tbl[bx]     ; 0 traduce il secondo byte
        mov   bl, [si+1]      ; 1 legge il primo byte per il prossimo ciclo
```

```
        nop                 ; 0 NOP per evitare il conflitto di banco
        mov     [si], al    ; 1 memorizza la traduzione del secondo byte
        add     si, 2       ; 0 fa avanzare il puntatore
        dec     cx          ; 1
        jnz     lbl         ; 0
                            ; —
                            ; 5 totale 2.5 per byte
```

La programmazione è una continua sfida

Si potrebbe pensare: "Utilizzando codice a 32 bit si potrebbero raddoppiare le prestazioni". Probabilmente questo non avviene ed ecco perché. Si torni per un attimo alle operazioni atomiche. Quale sarebbe la differenza utilizzando codice a 32 bit? Probabilmente nessuna. Un metodo che consentirebbe di accelerare ulteriormente il codice consiste nell'impiego di una tabella da 64 KB e nella traduzione di due byte per volta. Lo svantaggio principale di questo metodo consiste nel fatto che gli accessi casuali alla tabella da 64 KB riempirebbero costantemente la cache dei dati rallentando probabilmente il ciclo. La lettura di dati che non si trovano nella cache introduce un ritardo di tre cicli. Dunque anche se il ciclo di istruzioni potesse elaborare quattro byte in (teoricamente) cinque cicli di CPU, si dovrebbe comunque fare i conti con un ritardo di sei cicli.

Ecco dunque l'esercizio. Riscrivere il codice di traduzione in modo che richieda meno di due cicli per byte (su un Pentium) utilizzando meno di 1 KB di codice.

Verifiche di fattibilità

Per completezza ecco un altro metodo molto utilizzato per la conversione delle stringhe in lettere minuscole. Questo il metodo non è così versatile ma è molto più compatto in quanto esegue due confronti invece di utilizzare una tabella. Ecco il suo aspetto prima dell'ottimizzazione:

(Listato 14.15)

```
lbl:                        ; cicli
        lodsb               ; 2 legge un byte
        cmp     al, 'A'     ; 1
        jb      lbl3        ; 0 salta se è minore di 'A'
        cmp     al, 'Z'     ; 1
        ja      lbl2        ; 0 continua se è maggiore di 'Z'
        or      al, 20h     ; 1 conversione in minuscole
        mov     [si-1], al  ; 1 memorizza la traduzione
lbl2:
        loop lbl            ; 5
        jmp     done        ; —
lbl3:                       ; 11 cicli per byte
        test al, al         ; verifica del Null
        jnz     lbl2        ; torna indietro se non ci si trova a fine stringa
```

Ecco una sorpresa: confrontando questo codice con quello presentato all'inizio della sezione si scopre che questo metodo di confronto e salto si rivela più veloce rispetto al metodo non ottimizzato della tabella. Questo è dovuto al fatto che la coppia di istruzioni CMP/Jcc richiede un solo ciclo. Se si prova a misurare la velocità di questo codice e della versione ottimizzata illustrata di seguito, si scopre che il tempo di esecuzione dipende in modo notevole dai dati. Quando il sistema di previsione dei salti del Pentium esegue scelte corrette, una coppia di istruzioni CMP/Jcc o DEC/Jcc richiede un solo ciclo. Ecco una versione del codice che utilizza istruzioni semplici:

(Listato 14.16)

```
lbl:                        ; cicli
        mov   al, [si]      ; 1 legge un byte
        inc   si            ; 0
        cmp   al, 'A'       ; 1
        jb    lbl3          ; 0 salta se è minore di 'A'
        cmp   al, 'Z'       ; 1
        ja    lbl2          ; 0 continua se è maggiore di 'Z'
        or    al, 20h       ; 1 conversione in minuscole
        mov   [si-1], al    ; 1 memorizza la traduzione
lbl2:
        dec   cx            ; 1
        jnz   lbl           ; 0
        jmp   done          ; —
lbl3:                       ; 6 cicli per byte
        test al, al         ; verifica del Null
        jnz   lbl2          ; torna indietro se non ci si trova a fine stringa
```

Non si tratta un blocco di codice molto "elegante" ma esegue il proprio lavoro e ha il vantaggio di eliminare la tabella di 256 byte. I due salti interni di questo ciclo possono essere disposti in vari modi: il risultato migliore è di 3 cicli per byte mentre il peggiore è di 16 cicli per byte. Un ciclo per byte può essere eliminato elaborando due byte per ogni ciclo di istruzioni.

14.4 Ricerca di stringhe senza distinzione fra lettere maiuscole e lettere minuscole

L'ultimo problema di questo capitolo consiste nell'unire la ricerca e la conversione delle lettere in modo da eseguire una ricerca che non distingue fra lettere maiuscole e lettere minuscole. L'algoritmo di base prevede la scansione della stringa, la conversione di ogni carattere in lettere minuscole e successivamente il confronto. Quando viene trovata una corrispondenza, viene confrontata l'intera stringa convertendo ogni carattere prima del confronto. Per iniziare si proverà a fondere insieme le forme più semplici delle routine di scansione e di conversione. Si assumerà che la ricerca avvenga in una stringa di lunghezza nota.

(Listato 14.17)

```
; input: DI puntatore a una stringa di testo
;        CX lunghezza della stringa
;        SI puntatore ai caratteri da trovare
;           (in lettere minuscole)
```

```
        xor     bx, bx
        mov     dl, [si]        ;       carica il carattere di scansione
        inc     si              ;       fa avanzare il puntatore per il confronto

lbl:                            ; cicli
        mov     bl, [di]        ; 1 legge un carattere
        inc     di              ; 0 fa avanzare il puntatore
        mov     al, tbl[bx]     ; 2 lo traduce in minuscole (AGI)
        cmp     al, dl          ; 1 confronto con il carattere di scansione
        je      compare         ; 0
lbl2:
        dec     cx              ; 1
        jnz     lbl             ; 0
        jmp     no_match        ; —
                                ; 5 cicli per byte


compare:
        push si
        push di
m1:
        mov     al, [si]        ; 1 legge il carattere ricercato successivo
        inc     si              ; 0 fa avanzare il puntatore
        test al, al             ; 1 verifica del Null
        jz      exit            ; 0 termina a fine stringa
        mov     bl, [di]        ; 1 legge il carattere da convertire
        inc     di              ; 0 fa avanzare il puntatore
        mov     ah, tbl[bx]     ; 2 legge la conversione in minuscole (AGI)
        cmp     al, ah          ; 1 cerca una corrispondenza
        je      m1              ; 0 continua se sono uguali
exit:                           ; —
        pop     di              ; 6 cicli per carattere
        pop     si
        jne     lbl2            ; torna alla scansione se la ricerca non ha successo
```

Scansione di stringhe senza distinzione fra lettere minuscole e lettere minuscole

L'ottimizzazione della scansione e la disposizione delle parti del codice verranno considerate separatamente. La porzione di scansione è soggetta ai due problemi già visti precedentemente: un blocco AGI e un conflitto sui registri che impedisce l'accoppiamento delle istruzioni. Tali problemi verranno corretti come si è visto in precedenza elaborando un byte per ciclo. Per semplicità si assumerà che in CX sia contenuto un numero di byte pari:

(Listato 14.18)

```
        xor     bx, bx
        mov     dl, [si]        ; carica il carattere di scansione
        mov     bl, [di]        ; legge il primo carattere
        inc     di              ; fa avanzare il puntatore
```

```
        shr     cx, 1           ; regola il contatore del ciclo
        inc     si              ; fa avanzare il puntatore per il confronto

lbl:                            ; cicli
        mov     al, tbl[bx]     ; 1 traduzione in minuscole
        mov     bl, [di]        ; 0 legge il secondo carattere
        cmp     al, dl          ; 1 confronto con il carattere di scansione
        je      compare         ; 0
        mov     al, tbl[bx]     ; 1 traduce il secondo byte
        mov     bl, [di+1]      ; 0 legge il primo byte per il prossimo ciclo
        cmp     al, dl          ; 1 confronto con il carattere di scansione
        je      compare         ; 0
lbl2:
        add     di, 2           ; 1 fa avanzare il puntatore
        dec     cx              ; 0
        jnz     lbl             ; 1
                                ; —
                                ; 6 cicli, 3 per byte
```

Osservando questo codice è facile determinare le operazioni atomiche:

- lettura di un carattere dalla memoria;
- traduzione in lettere minuscole;
- confronto con il carattere ricercato;
- salto in caso di corrispondenza dei caratteri.

La condizione ottimale prevede due cicli di CPU per byte più il codice per la gestione del ciclo di istruzioni che non è possibile eliminare. Inoltre è possibile aggiungere agevolmente l'elaborazione di un maggior numero di byte per ciclo di istruzioni riducendo il numero di cicli di CPU a 2.67 (tre byte) o 2.5 (quattro byte).

Una soluzione che vale la pena di esplorare potrebbe essere la scansione della stringa due volte: una per le lettere maiuscole e una per le lettere minuscole (per ottenere le prestazioni ottimali la stringa nella quale viene eseguita la ricerca dovrebbe essere più piccola rispetto alle dimensioni della cache dei dati). Ma la conversione da maiuscole in minuscole aggiunge una sola operazione richiedendo solo 0.5 cicli di CPU per byte.

Confronto di stringhe senza distinzione fra lettere minuscole e lettere maiuscole

La seconda parte di questo esempio è il ciclo che confronta la stringa dopo che la scansione ha trovato una corrispondenza per quanto riguarda il primo carattere. Normalmente la velocità del confronto ha un impatto ridotto sulle prestazioni globali della ricerca. Tuttavia, se i dati contengono molte stringhe simili, la ricerca di una routine di confronto ottimizzata può risultare fondamentale. L'interfaccia con la routine di confronto è cambiata leggermente poiché è cambiata anche la routine di scansione. Ora si deve conservare il contenuto del registro BX e il registro DI può contenere valori diversi rispetto a quelli precedenti. Per ottenere in assoluto la massima veloci-

tà, si possono scrivere due diverse routine di confronto equivalenti, una da utilizzare nel caso di corrispondenza del primo byte e una per una corrispondenza sul secondo byte. Per semplificare la descrizione verrà utilizzata una sola routine:

(Listato 14.19)

```
compare:
        push  di
        push  si
        inc   di
        push  bx
m1:
        mov   ax, [si]        ; 1 legge il carattere ricercato successivo
        add   si, 2           ; 0 fa avanzare il puntatore
        mov   bl, [di]        ; 1 legge il carattere da convertire
        add   di, 2           ; 0 fa avanzare il puntatore
        test al, al           ; 1 verifica del Null
        jz    exit            ; 0 termina a fine stringa
        mov   dh, tbl[bx]     ; 1 legge la conversione in minuscole
        mov   bl, [di-1]      ; 0 legge il secondo carattere da convertire
        cmp   al, dh          ; 1 cerca una corrispondenza sul primo carattere
        jne   exit            ; 0 continua se sono uguali
        test ah, ah           ; 1 verifica del Null
        jz    exit            ; 0 termina a fine stringa
        mov   dh, tbl[bx]     ; 1 legge la conversione in minuscole
        cmp   ah, dh          ; 1 cerca una corrispondenza sul primo carattere
        je    m1              ; 0 continua se sono uguali
exit:                         ; —
        pop   bx              ; 8 cicli, 4 per carattere
        pop   si
        pop   di
        jne   lbl2            ; torna alla scansione se la ricerca non ha successo
```

Questo codice presenta un conflitto di registri che però non ha alcuna conseguenza poiché vi è un mezzo ciclo aggiuntivo non utilizzato. Ma è possibile ridurre il numero di cicli di CPU per ciclo di istruzioni osservando attentamente che non è necessario eseguire due confronti e due salti per carattere. Se ci si trova alla fine della sequenza di caratteri da ricercare (ovvero sul carattere nullo), questo non potrà corrispondere al carattere tradotto nella stringa in cui avviene la ricerca. Dunque si può specificare una stringa che non contenga il carattere nullo finale che può essere sostituito da qualsiasi altro valore sentinella, ad esempio il codice Control-Z (fine del file) oppure un -1. Se un'applicazione non può avere restrizioni di questo tipo, si può confrontare la lunghezza della stringa ricercata con il numero dei byte rimanenti nella stringa nella quale viene eseguita la ricerca. Se quest'ultimo valore è inferiore, non vi può essere alcuna corrispondenza.

(Listato 14.20)

```
compare
                              ; (verifica dei byte rimanenti)
        push di
        push si
        inc di                ; (inserire la versione per il 2o byte)
        push bx
        mov   bl, [di]        ; legge il carattere da convertire
```

```
m1:
        mov     ax, [si]        ; 1 legge il carattere ricercato successivo
        add     si, 2           ; 0 fa avanzare il puntatore
        mov     dh, tbl[bx]     ; 1 legge la conversione in minuscole
        mov     bl, [di+1]      ; 0 legge il secondo carattere da convertire
        cmp     al, dh          ; 1 cerca una corrispondenza sul primo carattere
        jne     exit            ; 0 esce se non vi è alcuna corrispondenza
        mov     dh, tbl[bx]     ; 1 legge la conversione in minuscole
        mov     bl, [di+2]      ; 0 legge il carattere da convertire
        add     di, 2           ; 1 fa avanzare il puntatore
        cmp     ah, dh          ; 0 cerca una corrispondenza sul secondo carattere
        je      m1              ; 1 continua se sono uguali
        exit:                   ; —
        pop     bx              ; 6 cicli, 3 per carattere
        pop     si
        pop     di
        test    al, al
        jz      done
        test    ah, ah
        jne     lbl2
done:
```

Le operazioni atomiche di un carattere nel precedente ciclo di confronto sono: leggere un carattere dalla stringa di testo;

- far avanzare il puntatore della stringa;
- leggere un carattere dalla sequenza di caratteri ricercata;
- far avanzare il puntatore della sequenza;
- tradurre il carattere della stringa di testo in lettere minuscole;
- confrontare i caratteri;
- saltare in caso di corrispondenza dei caratteri;.

Ora le operazioni sono diventate 7, 3 delle quali sono raddoppiate per elaborare due byte per ciclo. E nel ciclo di istruzioni vengono utilizzate 11 delle 12 posizioni per le istruzioni. Ma è possibile eseguire un'ulteriore ottimizzazione. Questo ciclo può essere utilizzato molto spesso ma il più delle volte terminerà dopo il confronto dei primi caratteri. Ecco una modifica che tiene in considerazione la possibilità che il ciclo termini dopo il confronto di un solo byte:

(Listato 14.21)

```
compare:                ; 1 byte 2 byte
        push    bx      ; 1      1
        mov     bl, [di] ; 0     0
        push    di      ; 1      1
        push    si      ; 0      0
        inc     di
m1:
        mov     ax, [si]        ; 1      1
        add     si, 2           ; 0      0
        mov     dh, tbl[bx]     ; 1      1
        mov     bl, [di+1]      ; 0      0
```

```
        cmp     al, dh          ; 1     1
        jne     exit1           ; 0     0
        mov     dh, tbl[bx]     ;       1
        mov     bl, [di+2]      ;       0
        add     di, 2           ;       1
        cmp     ah, dh          ;       0
        je      m1              ;       1
exit:                           ;
        pop     si              ;       1
        pop     di              ;       0
        pop     bx              ;       1
        test ah, ah     ;               0
        jne     lbl2            ;       1
        jmp     found   ;               0
exit1:                          ;       —
        pop     si              ; 1     11 cicli per 2 byte
        pop     di              ; 0
        pop     bx              ; 1
        test al, al             ; 0
        jne     lbl2            ; 1
found:                          ; —
                                ; 8 cicli per 1 byte
```

Conclusioni

Vi sono molte altre ottimizzazioni implementabili che hanno più a che fare con la sezione algoritmica del codice. Innanzitutto si potrebbe eseguire una scansione della stringa di testo ricercando un carattere poco frequente, piuttosto che il primo carattere. Un'estensione di questa idea, nel caso di ricerche senza distinzione fra lettere maiuscole e minuscole, consiste nel modificare le informazioni di frequenza per ricercare un carattere non alfabetico. Se si ricerca un carattere non alfabetico, non sarà necessario utilizzare la versione che non distingue le lettere maiuscole dalle lettere minuscole, ottenendo un ulteriore incremento di velocità.

Fino ad ora sono stati esplorati gli argomenti dell'utilizzo, ottimizzazione e accoppiamento delle istruzioni. Tuttavia la conoscenza ed esperienza si concentra su poche istruzioni ben note. A mano a mano che aumenta l'esperienza è possibile aumentare il numero di istruzioni utilizzate per poter scegliere strade meno battute.

Capitolo 15

Checksum e somme in precisione estesa

15.1 Somma in precisione estesa

Questo capitolo approfondisce il discorso dell'ottimizzazione dei cicli di istruzioni sul Pentium; tuttavia questi esempi riguardano solo operazioni di aritmetica intera. Per comprendere appieno gli elementi presentati è fondamentale lo studio dei due capitoli precedenti. Oltre a introdurre e ottimizzare nuovo codice, verrà presentata una metodologia per l'ottimizzazione superscalare che potrà essere applicata a varie situazioni.

Si partirà con un frammento di codice appartenente a un programma per l'8086. Tale programma è stato utilizzato sullo Space Shuttle per controllare alcuni esperimenti scientifici. Il progetto richiedeva l'esecuzione di test di memoria all'avvio, con continuità in background e su comando manuale.

Questo codice legge i byte contenuti in un blocco di memoria e li somma in un valore a 16 bit. Ecco l'aspetto del ciclo:

(Listato 15.1)

```
        xor     bx, bx
        mov     ah, 0
lbl:                    ; cicli
        lodsb           ; 2
        add     bx, ax  ; 1
        loop lbl        ; 5
                        ; —
                        ; 8 cicli per byte
```

Si tratta di un ciclo molto semplice e sarebbe difficile descriverlo in modo più compatto. L'obiettivo era quello di renderlo più veloce. Inoltre si deve ricercare il modo di ottenere prestazioni ottimali sul Pentium. Ecco come si può fare. Si può anche considerare questo problema come un esercizio e tentare di risolverlo da soli.

Fase 1

Come si è visto nei capitoli precedenti, il primo passo consiste nel trasformare tutte le istruzioni in istruzioni più semplici e accoppiabili. A tale scopo si può consultare il Capitolo 9 (Tabella 9.1). Queste istruzioni vengono in genere eseguite con il minor numero possibile di cicli di CPU e questo è molto importante. Ma è ancora più importante il fatto che queste istruzioni possano essere accoppiate con altre istruzioni. Se

vengono disposte correttamente, il numero di cicli di CPU può essere ridotto alla metà. Ecco come è possibile riscrivere questa porzione di codice utilizzando istruzioni semplici:

(Listato 15.2)

```
        xor     bx, bx
        mov     ah, 0
lbl:                            ; cicli
        mov     al, [si]        ; 1
        inc     si              ; 0
        add     bx, ax          ; 1
        dec     cx              ; 0
        jnz     lbl             ; 1
                                ; —
                                ; 3
```

Si è provato a espandere il codice aggiungendo altre istruzioni e in questo modo si è ottenuta una riduzione di cicli di CPU da 8 a 3. Ogni istruzione si è accoppiata a un'altra, tranne l'ultima. Ma questo non da grossi problemi. In fin dei conti la semplificazione è stata rapida ma è possibile fare di meglio.

Fase 2

Questo codice viene eseguito velocemente, ma osservando ciò che avviene, si nota che nei tre cicli di CPU vi sono sei possibili punti di esecuzione: uno non viene utilizzato per nulla e due si occupano della gestione del ciclo. Si tratta di un utilizzo del 50%; ora si può vedere ciò che accade eseguendo due somme per ciclo:

(Listato 15.3)

```
        xor     bx, bx
        mov     ah, 0
        mov     dh, 0
        shr     cx, 1           ; (divide il contatore per 2)
        jc      lbl_2           ; (salta se il numero è dispari)

lbl:                            ; cicli
        mov     al, [si]        ; 1
        inc     si              ; 0
        add     bx, ax          ; 1
lbl_2:
        mov     dl, [si]        ; 0 + 1 (AGI)
        inc     si              ; 1
        add     bx, dx          ; 0
        dec     cx              ; 1
        jnz     lbl             ; 0
                                ; —
                                ; 5 cicli, 2.5 per byte
```

Questo frammento di codice utilizza due cicli ma le prestazioni sono aumentate a 2.5 cicli per byte. Tuttavia si è introdotto un blocco AGI che deve essere rimosso. Si noti il metodo utilizzato per gestire il caso in cui debba essere elaborato un numero dispari di byte. Per chiarezza, i dettagli di quest'ultima ottimizzazione verranno de-

scritti più avanti. Inoltre si noti che i cicli vengono sempre interamente eseguiti senza alcun salto alle loro istruzioni interne.

Fase 3

In questo modo si è raddoppiato il lavoro eseguito da ogni ciclo di istruzioni ma il lavoro non è ancora finito. Si sarà probabilmente notato che SI viene incrementato due volte (provocando un blocco AGI). Questo problema può essere risolto molto facilmente:

(Listato 15.4)

```
lbl:                        ; cicli
        mov   al, [si]      ; 1
        add   bx, ax        ; 1 <— conflitto di registri, istruzione non accoppiabile
lbl_2:
        mov   dl, [si+1]    ; 0 <— bug
        add   si, 2         ; 1
        add   bx, dx        ; 0
        dec   cx            ; 1
        jnz   lbl           ; 0
                            ; —
                            ; 4 cicli, 2 per byte
```

Ma la soluzione non si presenta poi così semplice. Eliminando la prima istruzione INC SI si è creato un conflitto di registri e modificando la seconda lettura in memoria si è creato un bug (il bug ha a che fare con l'elaborazione di un numero dispari di byte). Il problema può essere risolto utilizzando un salto aggiuntivo all'interno del codice prima che il ciclo inizi a decrementare SI. Dunque si può provare a utilizzare un altro metodo:

(Listato 15.5)

```
lbl:                        ; cicli
        mov   dl, [si+1]    ; 1
        add   bx, dx        ; 0
lbl_2:
        mov   al, [si]      ; 1
        add   bx, ax        ; 1
        add   si, 2         ; 0 <— bug (SI deve essere decrementato
        dec   cx            ; 1 quando il contatore è dispari)
        jnz   lbl           ; 0
                            ; —
                            ; 4 cicli, 2 per byte
```

Come si può notare ora il codice è più confuso e non si è guadagnato nulla. Il registro SI viene incrementato di una unità nel primo ciclo nel caso in cui il numero di byte sia dispari.

Ecco un altro tentativo ma con il byte dispari sommato completamente all'esterno del ciclo che si sta cercando di ottimizzare. Se si ricerca la velocità, non vi è alcun motivo per costringere un ciclo a gestire il byte dispari. Se il risultato può essere ottenuto senza ridurre le prestazioni, tanto meglio, in caso contrario queste situazioni possono essere gestite all'esterno del ciclo, come nel seguente caso:

(Listato 15.6)

```
xor   bx, bx
      mov   ah, 0
      mov   dh, 0
      shr   cx, 1          ; divide il contatore per 2
      jnc   lbl            ; salta se è pari
      lodsb
      add   bx, ax
lbl:                       ; cicli
      mov   al, [si]       ; 1
      mov   dl, [si+1]     ; 0 + 1 conflitto di banco
      add   bx, ax         ; 1
      add   si, 2          ; 0
      add   bx, dx         ; 1
      dec   cx             ; 0
      jnz   lbl            ; 1
                           ; —
                           ; 5 cicli, 2.5 per byte
```

Anche questa versione richiede cinque cicli di CPU per ogni ciclo: vi è una sola posizione vuota, il codice richiede 2.5 cicli di CPU per byte in confronto agli otto iniziali e si potrebbe pensare che non sia possibile fare di meglio.

Fase 4

Purtroppo in questo modo si è ridotto il numero di istruzioni contenute nel ciclo (rispetto alla fase 2), ma la velocità rimane immutata e inoltre si continua a perdere un punto di esecuzione. Si potrebbe pensare di elaborare un terzo byte in ogni ciclo di istruzioni ma si continuerebbe a perdere un punto di esecuzione in quanto si aggiungerebbero altre due istruzioni. Osservando le istruzioni sembra che non sia possibile eliminarle e che tutte siano necessarie. Le uniche strade disponibili sembrano essere l'elaborazione di ulteriori byte per ciclo o l'utilizzo di codice a 32 bit.

Ma questa era solo la fase di "riscaldamento". Innanzitutto ci si può liberare del registro CX. In realtà non ha alcuno scopo se non il conteggio del numero dei cicli. Lo stesso lavoro può essere eseguito dal registro SP.

Si supponga che SI parta da 2000h e che CX sia 3000h. Basterà dunque modificarlo in modo che parta da un numero più alto meno un offset. L'idea è quella di far partire SI a un valore negativo in modo che il conteggio termini in corrispondenza del valore 0. SI viene modificato tramite uno scostamento in modo da ottenere il valore corretto per l'indirizzamento della memoria. La seguente tabella mostra un esempio:

	Originale	Modificato
SI	2000h	D000h
CX	3000h	-
Scostamento	0	+5000h

In pratica accadono le seguenti cose: il conteggio di SI parte da D000h e termina a 0, ovvero il ciclo termina quando SI raggiunge o supera il valore 0. L'indirizzo utilizzato per leggere il valore dalla memoria sarà D000h + 5000h ovvero 12000h; ma poiché

si stanno utilizzando registri a 16 bit, la prima cifra viene ignorata e dunque si ottiene il numero desiderato, ovvero 2000h. Gli stessi principi possono essere applicati anche a codice e registri a 32 bit.

Per chiarezza, il nuovo valore di scostamento verrà chiamato "scostamento modificato". Si assumerà che il byte dispari venga elaborato all'esterno del ciclo:

(Listato 15.7)

```
lbl:                            ; cicli
        mov     al, disp[si]    ; 1+1 <— Blocco AGI
        mov     dl, disp[si+1]  ; 0+1 <— conflitto di banco
        add     bx, ax          ; 1
        add     bx, dx          ; 1 <— istruzione non accoppiabile, conflitto di registri
        add     si, 2           ; 0 <— generazione dell'indirizzo
        jnc     lbl             ; 1
                                ; —
                                ; 6
```

Ora si è veramente creato qualche problema: si è incorsi in un conflitto di registri, un blocco AGI e un conflitto di banco. Le ultime due istruzioni del ciclo non possano essere facilmente modificate, anche se appena prima del salto condizionale può essere inserita un'istruzione che non modifica lo stato dei flag. Gli unici candidati per lo spostamento di istruzioni dipendono da SI. Le prime quattro istruzioni non possono essere disposte altrimenti senza provocare un altro conflitto di registri (se non si eseguono le somme prima degli spostamenti). L'introduzione di un nuovo registro ha anche lo scopo di eliminare il conflitto di registri nelle due somme. Le due somme verranno aggiunte alla fine del ciclo.

(Listato 15.8)

```
mov   al,    disp[si]
       mov   dl, disp[si+1]
lbl:                            ; cicli
       add bx, ax               ; 1
       add cx, dx               ; 0
       mov al, disp[si+2]       ; 1
       mov dl, disp[si+3]       ; 0+1 <— conflitto di banco
       add si, 2                ; 1
       jnc lbl                  ; 0
                                ; —
                                ; 4 cicli, 2 cicli per byte
```

Ci si preoccuperà del conflitto di banco in seguito. In questo modo si è ottenuto un ciclo di istruzioni che somma i byte in due cicli di CPU. Ma è possibile migliorarlo ulteriormente? Vale la pena di ricordare i principi delle operazioni atomiche. Per questo ciclo di istruzioni, le operazioni sono:

- lettura di un byte;
- somma del byte alla word;
- avanzamento del puntatore.

Tuttavia l'operazione sul puntatore è stata eliminata riunendola alle istruzioni di controllo del ciclo. Dunque il limite teorico è di 1 ciclo per byte e la procedura ottenuta può essere ulteriormente sviluppata.

Fase 5

Ancora non si è provato a utilizzare due registri per sommare 3 byte per ogni ciclo di istruzioni. Il limite in molti algoritmi è la natura "sparsa" dei registri disponibili sul Pentium. Ecco un modo per elaborare tre byte per ciclo:

(Listato 15.9)

```
        mov   al, [si+disp]
        mov   bl, [si+disp+1]
        mov   cl, [si+disp+2]
lbl:                          ; cicli
        add   di, ax          ; 1
        add   bp, bx          ; 0
        mov   al, [si+disp+3] ; 1
        mov   bl, [si+disp+4] ; 0+1 <— conflitto di banco
        add   di, cx          ; 1
        mov   cl, [si+disp+5] ; 0
        add   si, 3           ; 1
        jnc   lbl             ; 0
                              ; —
                              ; 5 cicli, 1.67 cicli per byte
```

Il calcolo del checksum è ora arrivato a 5 cicli di CPU per ogni ciclo di istruzioni ovvero a 1.67 cicli di CPU per ogni byte. Per chiarezza non sono stati mostrati i dettagli di gestione dei byte aggiuntivi quando il blocco di memoria non contiene un multiplo di 3 byte. Ma questo ciclo è molto più confuso dei precedenti. Dunque si proverà a sommare 4 byte per ciclo:

(Listato 15.10)

```
        mov   al, [si+disp]
        mov   dl, [si+disp+1]
lbl:                          ; cicli
        add   cx, ax          ; 1
        add   di, dx          ; 0
        mov   al, [si+disp+2] ; 1
        mov   dl, [si+disp+3] ; 0+1 <— conflitto di banco
        add   cx, ax          ; 1
        add   di, dx          ; 0
        mov   al, [si+disp+4] ; 1
        mov   dl, [si+disp+5] ; 0+1 <— conflitto di banco
        add   si, 4           ; 1
        jnc   lbl             ; 0
                              ; —
                              ; 7 cicli, 1.75 cicli per byte
```

Le prestazioni sono un po' inferiori ma il codice è meglio organizzato. Tuttavia occorre ancora liberarsi di qualche conflitto di banco.

Fase 6

I conflitti di banco sono gli ultimi ritardi che devono essere corretti. Ogni caso può essere diverso e dunque verranno presentati due metodi per eliminare questi conflitti.

Se possibile si dovrebbero riordinare le istruzioni. Spesso tale ottimizzazione sembra ovvia dopo averla vista all'opera. È un po' come vedere qualcun altro risolvere le parole crociate. Non sembra poi così difficile finché non si prova a risolverle da soli. In questo caso l'operazione non è poi così difficile:

(Listato 15.11)

```
        mov   al, [si+disp]
        mov   dl, [si+disp+1]
lbl:                                ; cicli
        add   cx, ax                ; 1
        mov   al, [si+disp+2]       ; 0
        add   di, dx                ; 1
        mov   dl, [si+disp+3]  ; 0
        add   cx, ax                ; 1
        mov   al, [si+disp+4]       ; 0
        add   di, dx                ; 1
        mov   dl, [si+disp+5]  ; 0
        add   si, 4                 ; 1
        jnc   lbl                   ; 0
                                    ; —
                                    ; 5 cicli, 1.25 cicli per byte
```

Chi avesse tentato a risolvere da solo questo problema avrebbe scoperto una delle tecniche più importanti nel riordinamento delle istruzioni: un registro che deve essere scritto ad ogni ciclo, in genere può essere scritto nell'istruzione accoppiata che segue l'istruzione in cui tale registro è stato letto.

Un secondo metodo per eliminare i conflitti di banco consiste nel predisporre un secondo flusso che legge i dati nel banco di memoria successivo. In questo caso si deve aggiungere il codice di gestione del secondo e quarto byte (oltre a tutti i byte dispari poiché il conteggio, nel caso di un ciclo per quattro byte, va da 0 a 3).

(Listato 15.12)

```
        mov   al, [si+disp]
        mov   dl, [si+disp+1]
lbl:                                ; cicli
        add   cx, ax                ; 1
        add   di, dx                ; 0
        mov   al, [si+disp+2]       ; 1
        mov   dl, [si+disp+7]  ; 0
        add   cx, ax                ; 1
        add   di, dx                ; 0
        mov   al, [si+disp+4]       ; 1
        mov   dl, [si+disp+9]  ; 0
        add   si, 4                 ; 1
        jnc   lbl                   ; 0
                                    ; —
                                    ; 5 cicli, 1.25 cicli per byte
```

Questo è tutto. Si sono risparmiati due registri e si è ottenuta una velocità teorica di 1.25 cicli di CPU per byte. Probabilmente ora il lettore può provare a creare cicli di istruzioni che leggano 6, 8 o un qualsiasi altro numero di byte per sinistro. Se vi è un metodo più veloce per risolvere questo problema, probabilmente affronta la situazione in modo completamente diverso.

Si sarà notata l'introduzione nella Fase 4 di uno "scostamento modificato" che però utilizzava una costante limitando la flessibilità del codice. Questo ciclo di istruzioni funziona solo con determinati valori di SI e quando il blocco ha dimensioni ben precise. Per correggere questa situazione è possibile utilizzare al posto dello scostamento modificato il registro BX, calcolandolo in modo opportuno:

(Listato 15.13)

```
                        ; gestisce innanzitutto il byte dispari
mov     bx, si
add     bx, cx          ; BX = SI + CX
mov     si, cx          ;
neg     si              ; SI = - CX

mov     al, [si+bx]
mov     dl, [si+bx+1]
lbl:                    ; cicli
add     cx, ax          ; 1
mov     al, [si+bx+2]   ; 0
add     di, dx          ; 1
mov     dl, [si+bx+3]   ; 0
add     cx, ax          ; 1
mov     al, [si+bx+4]   ; 0
add     di, dx          ; 1
mov     dl, [si+bx+5]   ; 0
add     si, 4           ; 1
jnc     lbl             ; 0
```

Eliminazione del ciclo

Naturalmente è possibile eliminare completamente questo ciclo. In questo modo si otterrebbero, teoricamente, le massime prestazioni possibili ovvero 1.0 cicli di CPU per byte. Innanzitutto questo risultato non è ottenibile e in secondo luogo, questa soluzione non sarebbe molto pratica, tranne che per blocchi di dati estremamente piccoli. Le massime prestazioni di 1.0 cicli di CPU per byte non possono essere raggiunte poiché il Pentium può accoppiare le istruzioni solo se queste sono già state eseguite dalla memoria cache (o nel caso delle poche istruzioni lunghe un solo byte).

Riepilogo

Nello sviluppo di questo esempio sono state tentate varie tecniche. Talvolta il procedimento seguito può essere sembrato come un procedere a tentoni nel labirinto dell'ottimizzazione. Ma questi problemi non prevedono metodi sicuri e ottimali per ottenere il successo. Quando si è in un labirinto, se si volta sempre verso destra (o verso sinistra), si incontrano numerose vie cieche ma alla fine si trova sicuramente l'uscita. Questo stesso principio può funzionare anche per l'ottimizzazione manuale dei programmi superscalari (assumendo che vi sia un'uscita per il "labirinto"). La Tabella 15.1 riassume le tecniche illustrate nelle fasi precedenti.

Tabella 15.1 Processo a sei fasi per l'ottimizzazione di programmi superscalari.

1.	Ridefinire il codice con istruzioni semplici.
2.	Utilizzare le due pipeline per flussi di dati indipendenti.
3.	Eliminare le operazioni sui puntatori.
4.	Eliminare, se possibile e vantaggioso, il contatore del ciclo di istruzioni.
5.	Duplicare quanto richiesto il codice del ciclo di istruzioni.
6.	Riordinare le istruzioni per eliminare i conflitti di banco e di registri.

Si possono utilizzare anche altri metodi. In particolare si deve elaborare un problema finché non si giunge a una via cieca, poi ritornare indietro e tentare un altro approccio. Si troveranno altre vie cieche. La domanda che ci si deve sempre porre è: "Questa via cieca è sufficientemente vicina all'uscita?"

Passi falsi

Se si utilizza una tecnica che non dà subito i risultati sperati, non si deve commettere l'errore di rinunciare troppo presto. Nella Fase 4, ad esempio, inizialmente si è persa un po' di velocità. Ecco un altro errore molto comune.

Tornando al codice della Fase 1 si può tentare di eliminare il contatore del ciclo.

(Listato 15.14)

```
        xor   bx, bx
        mov   ah, 0
lbl:                         ; cicli
        mov   al, disp[si]   ; 1+1 <— AGI
        add   bx, ax         ; 1 <— conflitto di registri, istruzione non accoppiabile
        inc   si             ; 0 <— generazione dell'indirizzo
        jnz   lbl            ; 1
```

Questo non migliora la situazione e anzi rallenta il ciclo! L'uso di un minor numero di registri aumenta le possibilità conflitti e di blocchi AGI. Naturalmente, un ciclo con più operazioni su ogni elemento potrebbe non avere questo problema.

15.1 Somma in precisione estesa

È interessante vedere il modo in cui i microprocessori 80x86 sono in grado di sommare (o sottrarre) valori binari composti da un qualsiasi numero di byte, word o dword:

(Listato 15.15)

```
        clc               ; cancella il carry per la prima somma
lbl:
        mov   al, [si]    ; legge un byte
        adc   [di], al    ; somma al byte il riporto della somma precedente
```

```
        inc  si         ; avanza al byte successivo
        inc  di         ; avanza al prossimo byte di destinazione
        loop lbl        ; continua fino alla fine
```

Studiando questo ciclo di istruzioni si scopriranno vari fatti interessanti. Questo ciclo funziona correttamente poiché le istruzioni INC e LOOP non modificano il flag carry. Il ciclo può essere facilmente modificato in modo da utilizzare word o dword. Può anche essere modificato in modo da utilizzare istruzioni che operano sulle stringhe. Questo deve essere stato il tipo di ciclo immaginato in fase di realizzazione dei microprocessori 8088/8086. Ecco una versione che opera su word:

(Listato 15.16)

```
clc
lbl:                          ; cicli
        lodsw                 ; 2 legge una word
        adc  [di], ax         ; 3 le somma il riporto
        inc  di               ; 1
        inc  di               ; 1 avanza alla word successiva
        loop lbl              ; 5 continua il ciclo
                              ; —
                              ; 12 cicli per word
```

Se si tenta di convertire questo ciclo in istruzioni semplici, possono sorgere dei problemi. La trasformazione di due INC in una ADD provoca la modifica del flag carry. Anche se la trasformazione dell'istruzione LOOP in istruzioni DEC/JNZ non modifica il flag carry, si può immaginare che debbano essere inserite altre istruzioni che modificano tale flag.

(Listato 15.17)

```
clc
lbl:                          ; cicli
        mov  ax, [si]         ; 1
        lea  si, [si+2]       ; 0 (LEA esegue la somma senza modificare i flag)
        mov  bx, [di]         ; 1
        lea  di, [di+2]       ; 0
        adc  ax, bx           ; 1
        mov  [di-2], ax       ; 1
        loop lbl              ; 5
                              ; —
                              ; 9 cicli per word
```

Questo ciclo è più veloce (9 cicli di CPU contro 12), ma decisamente non si tratta della soluzione ottimale in quanto utilizza solo 11 dei 18 punti di esecuzione disponibili. Si deve tentare qualche altro metodo che risparmia i flag utilizzati nel ciclo, ad esempio utilizzando le istruzioni PUSHF e POPF:

(Listato 15.18)

```
        clc
        pushf                 ; push dei flag prima del ciclo
lbl:                          ; cicli
        mov  ax, [si]         ; 1
        add  si, 2            ; 0
        mov  bx, [di]         ; 1
        add  di, 2            ; 0
        popf                  ; 6, PM = 4
```

```
        adc     ax, bx          ; 1
        mov     [di-2], ax      ; 1
        pushf                   ; 9, PM = 3
        dec     cx              ; 1
        jnz     lbl             ; 0
        popf                    ; (ripristina i flag dopo il ciclo)
                                ; —
                                ; 20 cicli
```

Ma in questo modo si rallenta il codice. Dunque è necessario tentare qualche altro metodo. Ad esempio, le istruzioni LAHF e SAHF sono molto più veloci rispetto alle istruzioni PUSHF/POPF:

(Listato 15.19)

```
clc
        lahf                    ; salva i flag prima del ciclo
lbl:                            ; cicli
        mov     dx, [si]        ; 1
        add     si, 2           ; 0
        mov     bx, [di]        ; 1
        add     di, 2           ; 0
        sahf                    ; 2
        adc     dx, bx          ; 1
        mov     [di-2], dx      ; 1
        lahf                    ; 2
        dec     cx              ; 1
        jnz     lbl             ; 0
                                ; —
                                ; 9 cicli
```

Ma utilizzando le istruzioni SAHF e LAHF si ottengono gli stessi problemi presentatisi con l'istruzione LOOP. Si può rinunciare a ricercare una soluzione elegante per questo problema ma in realtà vi è un'istruzione semplice che non utilizza il flag carry. Prima di procedere si può provare a ricercare tale istruzione nell'Appendice D.

(Listato 15.20)

```
xor     dx, dx
lbl:                            ; cicli
        mov     ax, [si]        ; 1
        add     si, 2           ; 0
        mov     bx, [di]        ; 1
        add     di, 2           ; 0
        rcr     dx, 1           ; 1
        adc     ax, bx          ; 1
        rcl     dx, 1           ; 1
        mov     [di-2], ax      ; 0
        dec     cx              ; 1
        jnz     lbl             ; 0
                                ; —
                                ; 6 cicli
```

Dunque le istruzioni RCR e RCL consentono di salvare e ripristinare il flag carry (in particolare entrambe le istruzioni eseguono simultaneamente il salvataggio e il ripristino). L'unico problema di questa porzione di codice è che le istruzioni di rota-

zione e l'istruzione ADC possono essere accoppiate solo nella pipe U. Ma questo provocherebbe un conflitto nel registro dei flag.

Ora si può provare a riorganizzare e/o ricodificare le istruzioni precedenti in modo che ogni istruzione possa essere accoppiata correttamente e che non vi siano conflitti nei registri.

(Listato 15.21)

```
xor   dx, dx
      mov   bx, [di]
lbl:                        ; cicli
      add   di, 2           ; 1
      mov   ax, [si]        ; 0
      rcr   dx, 1           ; 1
      lea   si, [si+2]      ; 0
      adc   ax, bx          ; 1
      mov   bx, [di]        ; 0
      rcl   dx, 1           ; 1
      mov   [di-2], ax      ; 0
      dec   cx              ; 1
      jnz   lbl             ; 0
                            ; —
                            ; 5 cicli
```

Questo ciclo di istruzioni richiede 5 cicli di CPU per word (ma può essere modificato in modo da richiedere 5 cicli per byte o per dword). Questo è probabilmente il problema di ottimizzazione più difficile presentato in questo manuale. È difficile che un qualsiasi compilatore sia in grado di generare codice tanto ottimizzato. Questo è il vantaggio di realizzare a mano il codice, specialmente quelle sezioni di codice la cui velocità particolarmente critica. Tuttavia questo codice è praticamente inutile.

Sì, inutile: quale applicazione potrebbe utilizzarlo? La maggior parte degli altri esempi presentati in questo manuale può essere utilizzata in varie situazioni. Questo esempio è stato scelto poiché mostra chiaramente il modo in cui è possibile conservare il flag carry in un ciclo. In altre parole queste tecniche possono essere utili anche se l'esempio specifico non lo è. Quante volte si può aver avuto bisogno di una somma di interi con precisione variabile? Se invece si ha bisogno di 6 word di precisione, si può partire dalle seguenti istruzioni:

(Listato 15.22)

```
                      ; cicli
mov   ax, [si]        ; 1
add   [di], ax        ; 3
mov   ax, [si+2]      ; 1
adc   [di+2], ax      ; 3
mov   ax, [si+4]      ; 1
adc   [di+4], ax      ; 3
```

Questo codice richiede quattro cicli per word ma può essere notevolmente ottimizzato. Innanzitutto si possono utilizzare più registri e riorganizzare le istruzioni:

(Listato 15.23)

```
                      ; cicli
mov   ax, [si]        ; 1
mov   bx, [si+2]      ; 0
add   [di], ax        ; 3
```

```
mov   ax, [si+4]     ; 0
adc   [di+2], bx     ; 3
adc   [di+4], ax     ; 3
```

Quindi si possono eliminare le istruzioni che richiedono 3 cicli di CPU:

(Listato 15.24)

```
                     ; cicli
mov   ax, [si]       ; 1
mov   bx, [di]       ; 0
add   ax, bx         ; 1
mov   cx, [si+2]     ; 0
mov   [di], ax       ; 1
mov   dx, [di+2]     ; 0
adc   cx, dx         ; 1
mov   ax, [si+4]     ; 0
mov   [di+2], cx     ; 1
mov   bx, [di+4]     ; 0
adc   ax, bx         ; 1
mov   [di+4], ax     ; 1
```

Prima di impegnarsi su una porzione di codice è sempre consigliabile chiedersi quante volte verrà eseguita. In questo problema di somma in precisione estesa, il codice finale non prevede alcun ciclo. Questo significa che le supposizioni di accoppiabilità potrebbero non essere corrette. Si deve sempre ricordare che per poter essere accoppiate, le istruzioni da eseguire nella pipe U devono essere lunghe un solo byte o devono essere già state eseguite dalla memoria cache. Se questa porzione di codice fa parte di una procedura richiamata molto frequentemente, probabilmente si fermerà a lungo nella cache e questo consentirà di eseguire somme a 48 bit con solo 7 cicli di CPU. Ogni volta che questa porzione di codice verrà reinserita nella memoria cache, richiederà invece 12 cicli poiché non sarà possibile accoppiarne le istruzioni. Il ciclo contenente l'istruzione LOOP richiederà invece da 16 a 21 cicli di CPU.

Parte quinta

ARGOMENTI AVANZATI

Capitolo 16

Operazioni matematiche in virgola mobile

In questo capitolo si parlerà dell'uso dell'unità in virgola mobile (FPU). Dopo aver trattato brevemente le operazioni di base di queste unità, si passerà ad affrontare l'argomento delle ottimizzazioni più avanzate. Chi non avesse mai programmato utilizzando istruzioni FPU, può rileggere la discussione riguardante la pipeline FPU presentata nel Capitolo 10.

Un particolare ringraziamento va a Harlan Stockman, il programmatore che ha scritto la maggior parte delle routine in virgola mobile e che ha eseguito le misurazioni contenute in questo capitolo.

Stockman ha dedicato molto tempo all'ottimizzazione di alcune routine di basso livello utili per scrivere applicazioni scientifiche. Tali routine sono state adottate per questo manuale ma rimangono ancora numerosi misteri irrisolti nell'utilizzo dell'unità in virgola mobile del Pentium. La speranza è che il lettore, dopo aver letto questo manuale, sia in grado di risolvere tali problemi.

16.1 Utilizzo dell'unità in virgola mobile

La famiglia delle unità in virgola mobile 80x87 include i chip 8087, 80287 e 80387 e le unità interne del 486 e del Pentium. In ogni caso si farà riferimento al coprocessore aritmetico (distinto o integrato nella CPU) con i nomi di unità in virgola mobile o unità FPU. Come si è detto nel Capitolo 10, l'unità FPU ha un'architettura a stack. Tale unità include le istruzioni per:

- caricare e memorizzare i dati (FLD, FST e così via);
- eseguire calcoli aritmetici (FADD, FMUL e così via);
- controllare il flusso del programma (FCOM, FTST e così via).

Tabella 16.1 Operazioni sullo stack in virgola mobile utilizzate nel Listato 16.1.

		;	st(0)	st(1)	st(2)	...1	st(7)
		;	-	-	-	-	-
fld	radius	;	2.0	-	-		-
fmul	st, st	;	4.0	-	-		-
ldpi		;	2.14+	4.0	-		-
fmul		;	12.56+	-	-		-
fld	side	;	3.0	12.56+	-		-
fmul	st, st	;	9.0	12.56+	-		-
fcom		;	9.0	12.56+	-		-
fstw	ax	;	9.0	12.56+	-		-
fstp	square	;	12.56+	-	-		-
fstp	circle	;	-	-	-		-
(radius dd2.0)							
(side dd3.0)							

Il programma di esempio presentato nel Listato 16.1 mostra alcune delle tecniche di programmazione di base dell'unità FPU.

La Tabella 16.1 mostra le istruzioni in virgola mobile utilizzate nel Listato 16.1 e il loro effetto sullo stack in virgola mobile. Come si ricorderà, lo stack FPU è costituito dal gruppo dei registri utilizzabili per la programmazione in virgola mobile. Può essere molto utile seguire l'andamento dello stack nel modo illustrato nella Tabella 16.1 specialmente quando si devono apprendere le tecniche di programmazione in virgola mobile e quando si devono eseguire operazioni complesse.

```
                      ; calcola l'area del cerchio          (Listato 16.1)
                      ; (pi * r * r)

fld     radius        ; carica il raggio
fmul    st, st        ; quadrato del raggio
fldpi                 ; carica pi
fmul                  ; moltiplica per ottenere l'area

; calcola l'area del quadrato

fld     side          ; carica il lato del quadrato
```

```
        fmul    st, st          ; quadrato del lato

        ; confronta le aree

        fcom                    ; confronto
        fstsw   ax              ; carica in AX lo stato dell'unità FPU

        ; memorizzazione dei risultati

        fstp    square          ; memorizza l'area del quadrato e la elimina dallo stack
        fstp    circle          ; memorizza l'area del cerchio e la elimina dallo stack

        ; salta sulla base del risultato del confronto

        sahf                    ; inserisce i flag FPU nei flag della CPU
        jp      fperr           ; il flag di parità è il flag C2 dell'unità FPU
                                ;  (quando è uguale a 1 si è verificato un errore)
        je      comp_equal
        jc      circle_larger
        jmp     square_larger

fperr:
        ...
comp_equal:
        ...
circle_larger:
        ...
square_larger:

        mov ah, 4ch
        int21h

main endp

.data

radius  dd 1.2          ; raggio del cerchio
side    dd 2.1          ; lato del quadrato
circle  dd 0            ; area del cerchio
square  dd 0            ; area del quadrato
```

16.2 **Ottimizzazione per l'utilizzo di matrici**

Nelle operazioni aritmetiche di base, l'unità FPU del Pentium è molto più veloce rispetto a quella del 486. Questo miglioramento è principalmente dovuto all'impiego di migliori algoritmi che riducono i tempi di esecuzione delle istruzioni; inoltre tali miglioramenti hanno il pregio di essere automatici. Tuttavia è possibile ottenere ulte-

riori miglioramenti tramite un'accorta programmazione delle istruzioni e tenendo in considerazione le possibilità di accoppiamento nella pipeline. I vantaggi di queste tecniche sono particolarmente evidenti nel caso dei cicli ripetuti molte volte. Un eccellente esempio che richiede molte dichiarazioni e un'attenta codifica è la moltiplicazione di due matrici di numeri in precisione doppia. Si partirà da un programma C che moltiplica due matrici in precisione doppia e crea una matrice prodotto (c [ ] [ ] = a [ ] [ ] * b [ ] [ ]).

```
void normal()                                        (Listato 16.2)
{
 int    i,j,k;
 for (i=0;i<N;i++){
        for (j=0;j<N;j++){
        c[i][j] = 0.0;
        for (k=0;k<N;k++){
        c[i][j] += a[i][k] * b[k][j];
        }
        }
 }
}
```

Questo algoritmo non si comporta troppo male se le matrici da moltiplicare rientrano entrambe nella cache dati del Pentium (8 KB). Ogni numero in doppia precisione richiede 8 byte e dunque nella cache possono entrare fino a 1024 numeri ovvero una matrice di 32 per 32 numeri. Ma i dati da manipolare sono contenuti in tre matrici (a[ ], b[ ] e c[ ]) più lo spazio per lo stack e le variabili. Per questo motivo le dimensioni massime delle matrici sono di circa 18 per 18 numeri. Nella maggior parte delle applicazioni scientifiche si tratta di dimensioni decisamente troppo ridotte. In questi casi in genere vengono gestite matrici molto più grandi, ad esempio di 500 per 500 elementi. Tre matrici di 500 per 500 elementi richiedono circa 6 MB di memoria e dunque è impossibile pensare di eseguire le operazioni accedendo sempre alla memoria cache.

Ecco dunque il problema evidenziato dal Listato 16.2. La memoria cache del Pentium è suddivisa in 128 linee, ognuna delle quali contiene 32 byte (ovvero 4 numeri in precisione doppia). Ogni volta che dalla memoria viene letto un elemento della matrice, il processore cerca di riempire una linea di cache utilizzando l'elemento letto e gli eletti che si trovano fisicamente vicini in memoria ad esso; l'architettura della memoria cache prevede un accesso sequenziale ai dati. Ma le dimensioni della memoria cache sono molto limitate e dunque se un programma legge elementi che in memoria sono piuttosto distanti fra loro, sarà difficile che venga riutilizzata una linea di cache poiché ogni volta sarà necessario sostituire i dati in essa contenuti.

L'accesso agli elementi di b[ ] avviene colonna per colonna e dunque ogni elemento è separato da 8 * n byte (n è il numero di elementi che costituiscono una riga della matrice). Ogni volta che si legge un elemento da b[ ], nella memoria cache vengono letti altri 3 elementi, assumendo che questi vengano ben presto utilizzati. Tuttavia questi elementi appartengono alle tre colonne successive e il programma non li utilizzerà se non quando sarà uscito al ciclo più esterno e a quel punto tale area della cache sarà stata sostituita da altri dati. La traduzione del ciclo interno in linguaggio assembler non migliora la situazione poiché il tempo necessario per leggere i dati nella CPU è

notevolmente superiore rispetto al tempo richiesto per eseguire la somma e la moltiplicazione.

Al contrario, i cicli interni dei due esempi successivi accedono alle matrici riga per riga e utilizzano la memoria cache in modo molto più efficiente.

L'esempio della moltiplicazione di matrici fornisce due lezioni fondamentali. Innanzitutto le routine dovrebbero essere scritte in modo che i cicli più interni operino, quando possibile, sulle righe e non sulle colonne di un array o almeno in modo da utilizzare i dati contenuti nella memoria cache. In secondo luogo, quando si deve ottimizzare una routine utilizzando il linguaggio assembler, ci si deve assicurare che l'algoritmo impiegato utilizzi la memoria cache in modo efficiente.

Alcune applicazioni si basano su matrici molto più piccole e in questo caso può essere utilizzato anche il Listato 16.2. L'impiego di una routine assembler può accelerare il codice contenuto nel Listato 16.2 ma non quanto i Listati 16.3 e 16.4. La Tabella 16.2 contiene le valutazioni dei vari metodi impiegati per scrivere una moltiplicazione fra matrici (la tabella include un confronto con il MIPS R4000).

```
void transpose()                                              (Listato 16.3)
{
        int i,j,k;
        double temp;
        for (i=0;i<N;i++){
                for (j=0;j<N;j++){
                bt[j][i] = b[i][j];
                }
        }
        for (i=0;i<N;i++){
                for (j=0;j<N;j++){
#ifndef asmloop
                temp = a[i][0] * bt[j][0];
                for (k=1;k<N;k++){
                temp += a[i][k] * bt[j][k];
                }
                c[i][j] = temp;
#else
                c[i][j] = ddot(N, a[i], bt[j]);
#endif
                }
        }
}

void reg_loops()                                              (Listato 16.4)
        {
        int i,j,k;
        double a_entry;
        for (i=0;i<N;i++){
                for (j=0;j<N;j++){
                c[i][j] = 0.0;
                }
        }
        for (i=0;i<N;i++){
```

```
                    for (k=0;k<N;k++){
#ifndef asmloop
                    a_entry = a[i][k];
                    for (j=0;j<N;j++){
                    c[i][j] += a_entry * b[k][j];
                    }
#else
                    daxpy(N, a[i]+k, b[k], c[i]);
#endif
                    }
          }
}
```

Nella ricerca dell'algoritmo più efficiente, si può notare che molti metodi sono implicitamente ottimizzati per CPU RISC e dunque potrebbero non essere adatti per il Pentium. Un esempio è costituito dal metodo warner() (righe 11 e 12 della Tabella 16.2). Nella versione pura realizzata in C, questo algoritmo non migliora sostanzialmente le prestazioni sul Pentium rispetto al semplice metodo reg_loops(). Al contrario il metodo warner() si rivela velocissimo su macchine RISC come il MIPS R4000. L'ultimo algoritmo copia gli elementi dell'array in numerose variabili temporanee per

Tabella 16.2 Tempo (in secondi) richiesto per la moltiplicazione di due matrici di 500 X 500 numeri in precisione doppia.

		Pentium 60 MHz	MIPS R4000 100 MHz
(1)	normal()	145.7	76.39
(2)	normal(), **a	186.2	128.6
(3)	normal(), ciclo interno in asm	121.6	-
(4)	transpose()	58.66	30.96
(5)	transpose(), **a	61.08	30.61
(6)	transpose(), asm ddot() nel ciclo interno	21.59	-
(7)	reg_loops()	43.44	37.00
(8)	reg_loops(), asm daxpy() nel ciclo interno	23.79	-
(9)	tiling()	53.28	40.29
(10)	tiling(), asm daxpy() nel ciclo interno	20.59	-
(11)	warner()	42.89	19.40
(12)	warner(), ciclo interno in assembler	18.13	-

evitare che il programma debba continuamente ricalcolare gli indirizzi nel ciclo più interno; questo approccio è ottimo nel caso di macchine RISC, in quanto il compilatore mantiene queste variabili nei registri. Ma il Pentium ha molti meno registri rispetto all'R4000 e dunque il compilatore C tende a tenere le variabili temporanee in memoria; questo è il motivo per il quale non è possibile ottenere vantaggi sensibili utilizzando il metodo warner() sul Pentium a meno che si impieghi una routine ottimizzata in linguaggio assembler per il ciclo interno, facendo attenzione a conservare le variabili temporanee nello stack dell'unità FPU.

16.3 In quale modo è meglio dichiarare gli array?

In genere si ritiene che sia meglio dichiarare di array bidimensionali come puntatori di puntatori a double, ovvero nel seguente modo:

```
double **a
```

piuttosto che nel seguente modo:

```
double a[500][500]
```

La seconda dichiarazione richiede infatti che per calcolare l'indirizzo di un elemento venga eseguita una moltiplicazione per la lunghezza della riga; poiché le moltiplicazioni fra interi sono relativamente lente (10 o 11 cicli di CPU su un Pentium o su un R4000), è ragionevole assumere che la dichiarazione double a[500][500] sia meno efficiente. Ma la seconda e la quinta riga della Tabella 16.2 mostrano un aspetto diverso. Per l'algoritmo normal la dichiarazione **a produce un codice notevolmente più lento sia sul Pentium che sull'R4000; su entrambi i processori i puntatori a[ ], b[ ] e c[ ] devono essere letti dalla memoria aggiungendo un ulteriore sovraccarico alla memoria cache. La perdita di velocità è invece più ridotta nel caso dell'algoritmo a trasposizione (riga 5 della Tabella 16.2) poiché per ogni prodotto deve essere letto un solo puntatore a ad a[ ], b[ ] e c[ ].

16.4 Ottimizzazione con il linguaggio assembler

Per ottenere maggiori prestazioni sul Pentium, si possono sostituire i cicli interni con due semplici routine assembler. La funzione ddot(), Listato 16.11, restituisce il prodotto dei vettori X e Y; la funzione daxpy(), Listato 16.5, calcola la somma dei vettori $ax + y$ dove a è uno scalare in modo da sostituire con il nuovo valore il vecchio valore di y. Sia ddot() che daxpy() elaborano n elementi per chiamata. Al termine, ogni funzione elaborerà n elementi in un ciclo di $n/4$ iterazioni, lasciando che gli ultimi n MOD 4 elementi vengano elaborati da un blocco apposito.

Queste funzioni assembler di basso livello sono ampiamente utilizzate e costituiscono il nucleo centrale di famosi pacchetti per l'algebra lineare come ad esempio LINPACK.

Dal punto di vista di un programmatore assembler, l'unità FPU del Pentium ha lo stesso funzionamento dei suoi predecessori 80x87 con alcune eccezioni. Innanzitutto è difficile ottenere un'esecuzione simultanea delle istruzioni intere ed FPU (vedere la descrizione fornita più avanti). In secondo luogo, con il Pentium è molto più importante evitare dipendenze fra i dati, ovvero operazioni FPU in sequenza che fanno riferimento alla stessa locazione di memoria e dunque mettono in stallo la pipeline. Infine l'istruzione FXCH può essere accoppiata con un'altra istruzione FPU (vedere la Tabella 10.6) e questo consente di eseguire tali istruzioni in 0 cicli.

Un'esecuzione simultanea di istruzioni FPU e intere sul Pentium è piuttosto difficile poiché molte delle più comuni istruzioni FPU sono così veloci che non presentano alcun "ritardo" utilizzabile per l'esecuzione di un'istruzione intera. Ad esempio, sul 486, l'istruzione FMUL richiede da 14 a 16 cicli mentre sul Pentium richiede da 1 a 3 cicli.

Il Listato 16.5 mostra un esempio della funzione daxpy() realizzata in assembler. Il listato mostra il numero dei cicli richiesti considerando che i valori vengano sempre caricati dalla memoria cache. Ovviamente questo non è possibile poiché ogni quattro letture di dati, un valore deve essere letto dalla memoria (un valore double di 8 byte in una linea di cache da 32 byte).

```
; void daxpy(int n, double *aptr, double *xptr, double *yptr)        (Listato 16.5)
;        calcola a*x[i] + y[i] e memorizza il risultato in y[]

        push   ebp
        mov    ebp, esp

        mov    ecx, dword ptr [ebp+8]      ; contatore
        mov    eax, dword ptr [ebp+12]     ; aptr
        fld    qword ptr [eax]
        mov    eax, dword ptr [ebp+16]     ; xptr
        mov    ebx, dword ptr [ebp+20]     ; yptr

lbl:                                       ; cicli
        fld    qword ptr [eax]             ; 1 caricamento di un double
        fmul   st, st(1)                   ; 1+2  moltiplicazione
        fadd   qword ptr [ebx]             ; 1+2  somma di un double
        fstp   qword ptr [ebx]             ; 2+1  memorizzazione e pop
        add    eax, 8                      ; 1 fa avanzare il puntatore al double successivo
        add    ebx, 8                      ; 0 fa avanzare il puntatore al double successivo
        loop   lbl                         ; 5 ciclo
                                           ; —
        fstp   st(0)                       ; 16 cicli per double (massimo accesso alla cache)
        pop    ebp
        ret
```

L'istruzione FMUL richiede un solo ciclo di CPU solo se le istruzioni seguenti non dipendono dal risultato della moltiplicazione; in caso contrario occorrerà considerare un ritardo di due cicli per completare l'istruzione. Lo stesso avviene nel caso dell'istruzione FADD. Inoltre, quando il risultato viene calcolato dall'istruzione precedente, FSTP richiede un ciclo aggiuntivo. Per ottimizzare il codice FPU occorre

eliminare tutti questi ritardi. Si partirà dal Listato 16.6 aprendo il ciclo in modo da eseguire due operazioni:

```
lbl:                                ; cicli              (Listato 16.6)
        fld     qword ptr [eax]     ; 1
        fmul    st, st(1)           ; 1+2
        fadd    qword ptr [ebx]     ; 1+2
        fstp    qword ptr [ebx]     ; 2+1

        fld     qword ptr [eax+8]   ; 1
        fmul    st, st(1)           ; 1+2
        fadd    qword ptr [ebx+8]   ; 1+2
        fstp    qword ptr [ebx+8]   ; 2+1

        add     eax, 16             ; 1
        add     ebx, 16             ; 0
        dec     ecx                 ; 1
        jnz     lbl                 ; 0
                                    ; —
                                    ; 22 cicli, 11 per double
```

La corretta disposizione delle istruzioni FPU è un po' più complessa rispetto a quanto si è visto in precedenza per le istruzioni intere. Questo è dovuto principalmente al fatto che l'unità FPU opera con una serie di registri disposti in modo da formare uno stack. Un modo per disporre le istruzioni consiste nel lasciare semplicemente che i risultati si aggiungano l'uno all'altro sullo stack per poi estrarli al termine delle operazioni. Questo elimina il ritardo introdotto dall'istruzione FADD:

```
lbl:                                ; cicli              (Listato 16.7)
        fld     qword ptr [eax]     ; 1
        fmul    st, st(1)           ; 1+2
        fadd    qword ptr [ebx]     ; 1

        fld     qword ptr [eax+8]   ; 1
        fmul    st, st(2)           ; 1+2
        fadd    qword ptr [ebx+8]   ; 1+2

        fstp    qword ptr [ebx+8]   ; 2+1
        fstp    qword ptr [ebx]     ; 2

        add     eax, 16             ; 1
        add     ebx, 16             ; 0
        dec     ecx                 ; 1
        jnz     lbl                 0
                                    ; —
                                    ; 19 cicli, 9.5 cicli per double
```

Grazie all'accoppiabilità dell'istruzione FXCH è possibile eliminare il ritardo di FSTP cambiando i primi due elementi dello stack:

```
lbl:                                ; cicli              (Listato 16.8)
        fld     qword ptr [eax]     ; 1
```

```
        fmul    st, st(1)               ; 1+2
        fadd    qword ptr [ebx]         ; 1

        fld     qword ptr [eax+8]       ; 1
        fmul    st, st(2)               ; 1+2
        fadd    qword ptr [ebx+8]       ; 1

        fxch    st(1)                   ; 0
        fstp    qword ptr [ebx]         ; 2
        fstp    qword ptr [ebx+8]       ; 2+1

        add     eax, 16                 ; 1
        add     ebx, 16                 ; 0
        dec     ecx                     ; 1
        jnz     lbl                     ; 0
                                        ; —
                                        ; 16 cicli, 8 cicli per double
```

Tuttavia, eseguendo questa prima metà di ogni operazione, la seconda metà elimina tutti i ritardi:

(Listato 16.9)

```
lbl:                                    ; cicli
        fld     qword ptr [eax]         ; 1
        fmul    st, st(1)               ; 1
        fld     qword ptr [eax+8]       ; 1
        fmul    st, st(2)               ; 1
        fxch    st(1)                   ; 0

        fadd    qword ptr [ebx]         ; 1
        fxch    st(1)                   ; 0
        fadd    qword ptr [ebx+8]       ; 1
        fxch    st(1)                   ; 0
        fstp    qword ptr [ebx]         ; 2
        fstp    qword ptr [ebx+8]       ; 2

        add     eax, 16                 ; 1
        add     ebx, 16                 ; 0
        dec     ecx                     ; 1
        jnz     lbl                     ; 0
                                        ; —
                                        ; 12 cicli, 6 per double
```

Estendendo il ciclo a quattro operazioni per iterazione se ne migliorano leggermente le prestazioni (vedere il Listato 16.10).

(Listato 16.10)

```
lbl:                                    ; cicli
        fld     qword ptr [eax]         ; 1
        fmul    st, st(1)               ; 1
        fld     qword ptr [eax+8]       ; 1
        fmul    st, st(2)               ; 1
        fxch    st(1)                   ; 0
```

```
fadd    qword ptr [ebx]       ; 1
fxch    st(1)                 ; 0
fadd    qword ptr [ebx+8]     ; 1
fxch    st(1)                 ; 0
fstp    qword ptr [ebx]       ; 2
fstp    qword ptr [ebx+8]     ; 2

fld     qword ptr [eax+16]    ; 1
fmul    st, st(1)             ; 1
fld     qword ptr [eax+24]    ; 1
fmul    st, st(2)             ; 1
fxch    st(1)                 ; 0

fadd    qword ptr [ebx+16]    ; 1
fxch    st(1)                 ; 0
fadd    qword ptr [ebx+24]    ; 1
fxch    st(1)                 ; 0
fstp    qword ptr [ebx+16]    ; 2
fstp    qword ptr [ebx+24]    ; 2

add     eax, 32               ; 1
add     ebx, 32               ; 0
dec     cxx                   ; 1
jnz     lbl                   ; 0
                              ; —
                              ; 22 cicli, 5.5 per double
```

Così come avveniva nel caso delle istruzioni intere, osservando le operazioni atomiche è possibile conoscere, approssimativamente, le migliori prestazioni che è possibile ottenere:

- caricamento di un numero;
- moltiplicazione;
- somma;
- memorizzazione (2).

In questo caso sembra che, escludendo le istruzioni di gestione del ciclo, si possa ottenere un risultato di 5 cicli di CPU per elemento.

Anche la funzione daxpy(), così come la funzione ddot() può essere ottimizzata. Quella che segue è una tipica implementazione in linguaggio assembler della funzione ddot() (vedere il Listato 16.11).

```
; double ddot(int n, double *xptr, double *yptr)                    (Listato 16.11)
;       calcola il prodotto di due vettori (righe)
;       restituisce il risultato in edx:eax e __fac

    push  ebp
    mov   ebp, esp

    mov   ecx, dword ptr [ebp+8]        ; contatore
    mov   eax, dword ptr [ebp+12]       ; xptr
```

```
        mov   ebx, dword ptr [ebp+16] ;          yptr
        fldz

lbl                                ; cicli
        fld   qword ptr [eax]      ; 1    carica double
        fmul  qword ptr [ebx]      ; 1+2  moltiplica
        fadd                       ; 1    somma
        add   eax, 8               ; 1    fa avanzare il puntatore al double successivo
        add   ebx, 8               ; 0    fa avanzare il puntatore al double successivo
        loop  lbl                  ; 5    ciclo
                                   ; —
        fstp  __fac                ; 11 cicli per double (massimo accesso alla cache)
        mov   eax, dword ptr __fac
        mov   edx, dword ptr __fac+4
        pop   ebp
        ret
```

La funzione ddot() restituisce il risultato in una variabile globale chiamata __fac ed anche nei registri EDX:EAX. I vari compilatori C possono seguire convenzioni differenti a proposito del modo in cui restituire il risultato di una funzione. Potrebbe non essere necessario caricare nuovamente il risultato nei registri EDX:EAX, ma l'istruzione FSTP richiede sempre di memorizzare il risultato in una variabile globale o in una variabile locale dello stack. Alcuni compilatori richiedono che il risultato della funzione rimanga nello stack FPU alla posizione st(0).

Il primo passo per ottimizzare questo codice consiste nell'impiego di istruzioni intere semplici e nel massimo sfruttamento delle due pipeline.

(Listato 16.12)

```
lbl:                               ; cicli
        fld   qword ptr [eax]      ; 1    carica A1
        fmul  qword ptr [ebx]      ; 1+2  moltiplica A1 * A2
        fadd                       ; 1    somma sum+A1*A2
        fld   qword ptr [eax+8]    ; 1    carica B1
        fmul  qword ptr [ebx+8]    ; 1+2  moltiplica B1 * B2
        fadd                       ; 1    somma sum+B1*B2
        add   eax, 16              ; 1
        add   ebx, 16              ; 0
        dec   ecx                  ; 1
        jnz   lbl                  ; 0
                                   ; —
                                   ; 12 cicli, 6 per double
```

In questo modo si aumentano le prestazioni ma non si risolve il problema dei ritardi aggiuntivi provocati dall'uso del risultato di FMUL nell'istruzione successiva. Occorre sempre ricordare che anche il Pentium è in grado di eseguire una sola istruzione FPU per volta (ad eccezione solo dell'istruzione FXCH). L'obiettivo nella programmazione di codice in virgola mobile per il Pentium consiste nell'ottimizzare il funzionamento della pipeline per eliminare le dipendenze fra i dati. L'utilizzo delle due pipeline si applica solo alle istruzioni intere. Ma aprendo il ciclo di istruzioni in modo che operi su due o più gruppi di dati si può ottenere un grande vantaggio poiché i ritardi delle operazioni FPU vengono eliminati dal fatto che i risultati di un'operazione vengono richiesti solo con uno o due cicli di CPU di ritardo.

Il Listato 16.13 mostra un semplice metodo che consente di eliminare uno dei due ritardi.

(Listato 16.13)

```
        jmp     lbl2                    ; cicli
lbl:
        fadd                            ; 1     somma sum+A1*A2

lbl2:
        fld     qword ptr [eax]         ; 1     carica A1
        fmul    qword ptr [ebx]         ; 1     moltiplica A1 * A2
        fld     qword ptr [eax+8]       ; 1     carica B1
        fmul    qword ptr [ebx+8]       ; 1+2   moltiplica B1 * B2
        fadd                            ; 1     somma sum+B1+B2
        add     eax, 16                 ; 1
        add     ebx, 16                 ; 0
        dec     ecx                     ; 1
        jnz     lbl                     ; 0
        fadd                            ; —
                                        ; 10 cicli, 5 per double
```

Ritardando, per ogni iterazione, la seconda istruzione FADD fino all'inizio del ciclo successivo è possibile eseguire le istruzioni di gestione di tale ciclo di istruzioni durante i due cicli di CPU necessari perché diventi disponibile il primo risultato di FADD. Questa tecnica è vantaggiosa poiché non genera cicli aggiuntivi neanche sui precedenti microprocessori.

Il Listato 16.14 utilizza l'istruzione FXCH per eliminare il ritardo sulla prima istruzione FADD.

(Listato 16.14)

```
        jmp     lbl2                    ; cicli
lbl:
        fadd                            ; 1     somma sum+B1*B2

lbl2:
        fld     qword ptr [eax]         ; 1     carica A1
        fmul    qword ptr [ebx]         ; 1     moltiplica A1*A2
        fld     qword ptr [eax+8]       ; 1     carica B1
        fmul    qword ptr [ebx+8]       ; 1     moltiplica B1*B2
        fxch                            ; 0     scambia st(0) e st(1)
        faddp   st(2), st               ; 1     somma sum+A1*A2
        add     eax, 16                 ; 1
        add     ebx, 16                 ; 0
        dec     ecx                     ; 1
        jnz     lbl                     ; 0
        fadd                            ; —
                                        ; 8 cicli, 4 per double
```

Il Listato 16.14 utilizza le istruzioni FXCH e FADDP per cambiare l'ordine in cui vengono eseguite le somme. La Tabella 16.3 mostra i dettagli dello stack mentre vengono eseguite queste operazioni.

Infine il ciclo di istruzioni può essere aperto in modo da operare su quattro gruppi di dati ad ogni iterazione.

Tabella 16.3 Operazioni in virgola mobile sullo stack per il Listato 16.14.

	;	st(0)	st(1)	st(2)	...	st(7)
	;	sum	-	-	-	-
fld	;	A1	sum	-	-	-
fmul	;	Al*A2	sum	-	-	-
fld	;	B1	A1*A2	sum	-	-
fmul	;	B1*B2	A1*A2	sum	-	-
fxch	;	A1*A2	B1*B2	sum	-	-
faddp st(2), st	;	B1*B2	sum+A1*A2	-	-	
...						
fadd	;	sum+A1*A2 +B1*B2	-	-	-	-

(Listato 16.15)

```
        jmp     lbl2              ; cicli
lbl:
        fadd                      ; 1    somma sum+D1*D2
lbl2:
        fld     qword ptr [si]    ; 1    carica A1
        fmul    qword ptr [di]    ; 1    moltiplica A1*A2
        fld     qword ptr [si+8]  ; 1    carica B1
        fmul    qword ptr [si+8]  ; 1    moltiplica B1*B2
        fxch                      ; 0    scambia st(0) e st(1)
        faddp   st(2), st         ; 1    somma sum+A1*A2

        fld     qword ptr [si+16] ; 1    carica C1
        fmul    qword ptr [di+16] ; 1    moltiplica C1*C2
        fxch                      ; 0    scambia st(0) e st(1)
        faddp   st(2), st         ; 1    somma sum+B1*B2

        fld     qword ptr [si+24] ; 1    carica D1
        fmul    qword ptr [di+24] ; 1    moltiplica D1*D2
        fxch                      ; 0    scambia st(0) e st(1)
        faddp   st(2), st         ; 1    somma sum+C1*C2

        add     si, 32            ; 1
        add     di, 32            ; 0
        dec     cx                ; 1
        jnz     lbl               ; 0
        fadd                      ; —
lbl3:                             ; 14 cicli, 3.5 per double
```

La Tabella 16.4 mostra i dettagli dello stack mentre vengono eseguite le operazioni del Listato 16.15.

Come nel caso delle istruzioni intere e dell'esempio della funzione daxpy(), osservando le operazioni atomiche è possibile conoscere, approssimativamente, le migliori prestazioni che è possibile ottenere:

- caricamento di un numero;
- moltiplicazione;
- somma.

Tabella 16.4 Operazioni in virgola mobile sullo stack per il Listato 16.15.

		st(0)	st(l)	st(2)	...	st(7)
	;	sum	-	-		-
fld	;	A1	sum	-		-
fmul		A1*A2	sum	-		-
fld	;	B1	A1*A2	sum		-
fmul	;	B1*B2	A1*A2	sum		-
fxch	;	A1*A2	B1*B2	sum		-
faddp st(2), st	;	B1*B2	sum+A1*A2	-		-
fld	;	C1	B1*B2	sum+A1*A2		-
fmul	;	C1*C2	B1*B2	sum+A1*A2		-
fxch	;	B1*B2	C1+C2	sum+A1*A2		-
faddp st(2), st	;	C1*C2	sum+A1*A2 +B1*B2	-		-
fld	;		C1*C2	sum+A1*A2 +B1*B2		-
fmul	;	D1*D2	C1*C2	sum+A1*A2 +B1*B2		-
fxch	;	C1*C2	D1*D2	sum+A1*A2 +B1*B2		-
faddp st(2), st	;	D1*D2	sum+A1*A2 +B1*B2 +C1*C2	-		-
...						
fadd	;	sum+A1*A2 +B1*B2 +C1*C2 +D1*D2	-	-		-

In questo caso è possibile ottenere 3 cicli di CPU, escludendo le istruzioni di gestione del ciclo che portano il risultato finale a 3.5 cicli di CPU.

I risultati presentati nella Tabella 16.2 mostrano che le prestazioni delle applicazioni C o C++ possono essere notevolmente migliorate impiegando piccole porzioni di codice assembler ottimizzato. Sfruttando adeguatamente le potenzialità del Pentium non è difficile ottenere miglioramenti del 100%. E occorre notare che il confronto avviene nei confronti di un algoritmo che utilizza al meglio la memoria cache disponibile nel Pentium. Nel prossimo capitolo verranno descritti i metodi di interfacciamento del linguaggio assembler con il C e il C++.

Capitolo 17

Interfacciamento con il C

In questo capitolo verranno discusse varie tecniche per includere routine assembler all'interno di programmi C o C++. In particolare verranno discussi tre metodi:

- istruzioni assembler in-line;
- moduli assembler e C da unire tramite linking;
- moduli assembler e C da unire tramite linking con fastcall.

Saranno descritti i requisiti di interfacciamento per ognuno di questi metodi, correlandoli di esempi e considerandone vantaggi e svantaggi. Infine si vedrà come è possibile misurare le prestazioni del codice scritto utilizzando il timer descritto nel Capitolo 12.

17.1 Linguaggio assembler in-line

Probabilmente il modo più facile per includere codice assembler in un programma C o C++ consiste nell'impiegare codice assembler in-line. Si tratta di una possibilità messa a disposizione dal compilatore C che consente di inserire il codice assembler direttamente all'interno di un programma C o C++. Il Listato 17.1 mostra un esempio funzionante con il C/C++ Microsoft 7.0 e successivi (incluso il Visual C++):

(Listato 17.1)

```
int add2c( int x, int y)    /* esempio di funzione C */
{
 return x+y;
}


int add2a( int x, int y)    /* stessa funzione realizzata in assembler */
{
 __asm{
         mov ax, y
         add ax, x
        }
}
```

Ma la funzione assembler può essere scritta anche nel seguente modo:

(Listato 17.2)

```
int add2a( int x, int y)
{
 __asm  mov ax, y
 __asm  add ax, x
}
```

Per il Borland C/C++ si dovrà utilizzare la seguente forma:

(Listato 17.3)

```
#pragma  inline

int add2( int x, int y)
{
 asm  mov ax, y
 asm  add ax, x
}
```

L'utilizzo di direttive #ifdef consente di compilare lo stesso codice con entrambi i compilatori:

```
#ifdef (_MSC_VER >= 700)
#   define asm __asm
#endif
```

Tuttavia, per chiarezza, verrà utilizzata la struttura a blocchi del Microsoft C/C++ 7.0. Le istruzioni __asm o asm possono essere inserite direttamente nella riga di codice in cui sono richieste.

La scelta tra codice assembler in-line e codice assembler in un file distinto è più che altro questione di stile e di preferenza personale. Tuttavia, entrambi i metodi presentano vantaggi e svantaggi. I vantaggi del codice assembler in-line sono:
è rapido e semplice da usare;

- non richiede l'impiego di un assembler distinto (ad esempio MASM o TASM);
- consente di utilizzare con facilità le variabili e le funzioni del C;
- il controllo delle versioni risulta più semplice;
- nella stessa funzione è possibile utilizzare insieme codice C e assembler;
- non è richiesta la conoscenza delle convenzioni di nome del C++;
- non è richiesta una grande conoscenza delle convenzioni di chiamata.

Gli svantaggi del codice assembler in-line sono:

- non ha la potenza di un vero assembler;
- non è possibile controllare la segmentazione;
- non è possibile sfruttare tutte le potenzialità delle dichiarazioni dei dati e delle macro.

Esempio di codice assembler in-line

In questo esempio si proverà a sostituire la funzione strcpy() contenuta nella libreria standard del C con il codice di copia di una stringa presentato nel Capitolo 13. Naturalmente si assumerà che questo codice sia più veloce ma le verifiche verranno eseguite in seguito. Ecco come è possibile utilizzare la funzione strcpy() in un semplice programma:

(Listato 17.4)

```
#include <string.h>
void main()
{
        char buffer1[80] = "Stringa di esempio";
        char buffer2[80];
        strcpy(buffer2, buffer1);
        printf("stringa 2 = %s\n", buffer2);
}
```

Per modificare il programma si deve aggiungere una nuova funzione strcpy():

(Listato 17.5)

```
#include <string.h>
void main()
{
 char buffer1[80] = "Stringa di esempio";
 char buffer2[80];

 strcpy(buffer2, buffer1);
 printf("stringa 2 = %s\n", buffer2);
}
char *strcpy(char *string1, const char *string2)
{
 __asm{
        mov    di, string1
        mov    si, string2
        lbl:
        mov    ax, [si]
        add    si, 2
        cmp    al, 0h
        je     exit2
        mov    [di], ax
        add    di, 2
        cmp    ah, 0h
        jne    lbl
        exit:
        jmp    exit3
        exit2:
        mov    [di], al
        exit3:
        mov    ax, string1      ; restituisce il puntatore
  }
}
```

17.2 Linking di moduli distinti

Mantenendo il codice assembler distinto rispetto al codice C o C++ è più facile ottenere un completo controllo delle versioni. Molte volte questo è necessario poiché si intende sfruttare tutte le potenzialità dell'assembler. In entrambi i casi è piuttosto facile scrivere il codice e assemblarlo per poi eseguire il linking con i moduli oggetto prodotti dal compilatore C o C++. Una delle difficoltà è costituita dal passaggio dei parametri dal codice C o C++ alla routine assembler. Per questo motivo è necessario conoscere le convenzioni di chiamata. Nel caso dell'assembler in-line tale conoscenza non era necessaria.

La realizzazione di una procedura assembler richiamabile da un programma scritto in un linguaggio di alto livello prevede numerose fasi. Alcune di queste fasi possono non essere necessarie in tutte le situazioni ma si può utilizzare questo elenco come una guida:

- dichiarare il nome della procedura;
- predisporre lo stack frame;
- allocare, se necessario, lo spazio locale sullo stack;
- salvare i registri utilizzati;
- caricare o accedere ai parametri contenuti nello stack;
- eseguire le operazioni per le quali si è realizzata la funzione;
- determinare l'eventuale valore della funzione;
- ripristinare i registri;
- deallocare l'eventuale spazio locale dello stack;
- ripristinare lo stack;
- uscire dalla procedura.

Il capitolo discuterà questi argomenti in dettaglio, ma innanzitutto occorre sapere il modo in cui i vari linguaggi gestiscono alcune di queste fasi.

Convenzioni di chiamata

I linguaggi utilizzano nomi differenti per le subroutine, le procedure e le funzioni. In C viene utilizzato il termine funzione e in C++ vi sono funzioni e metodi. In altri linguaggi vi è una differenziazione fra funzioni che restituiscono un risultato e altre routine che non restituiscono alcun risultato. Generalmente vengono utilizzati i termini routine, subroutine e procedura. Le funzioni sono dunque routine, subroutine e procedure che restituiscono un valore.

I linguaggi di alto livello passano i parametri alle subroutine e alle funzioni utilizzando lo stack di sistema. Per riuscire a scrivere una funzione che accetta parametri in input, occorre sapere alcune cose a proposito del modo in cui i vari linguaggi inseriscono ed estraggono elementi dallo stack. La Tabella 17.1 mostra le convenzioni utilizzate da alcuni linguaggi.

L'ordine dei parametri fa riferimento alla sequenza in cui gli elementi vengono inseriti nello stack. In C e C++, questi vengono inseriti in ordine inverso (da destra

verso sinistra). Con reinizializzazione dello stack si fa riferimento all'operazione di riposizionamento del puntatore allo stack nella sua posizione originaria. In C e C++ è la funzione chiamante che inserisce i parametri nello stack, richiama la funzione chiamata e quindi ripristina il puntatore allo stack riportandolo nella posizione iniziale. In altri linguaggi l'operazione di ripristino deve essere eseguita dalla funzione chiamata tramite un'istruzione RET # dove # è il numero di dati inseriti nello stack dalla funzione chiamante. Le convenzioni dei nomi specificano che le funzioni C devono essere dichiarate con un carattere di sottolineatura iniziale e che per i nomi delle funzioni si fa differenza fra lettere maiuscole e minuscole.

I parametri possono essere inseriti nello stack per valore o per indirizzo. La Tabella 17.1 mostra le convenzioni generali utilizzate in ogni linguaggio. Ogni tipo di dati, in ogni linguaggio, può avere proprie convenzioni di chiamata. Con passaggio per valore si intende l'inserimento nello stack dell'effettivo valore di una variabile. Con passaggio per indirizzo si intende l'inserimento nello stack di un puntatore alla variabile. I puntatori possono essere costituiti dal solo offset (puntatori near) o dal segmento e dall'offset (puntatori far). Quando viene passato il solo offset, si assume l'utilizzo del segmento dati (DS). I vantaggi del passaggio per valore sono la velocità (per elementi di piccole dimensioni), la compattezza del codice e la possibilità di evitare che la subroutine modifichi il valore della variabile. Lo svantaggio è ovviamente dovuto al fatto che la subroutine non può modificare il valore. I vantaggi del passaggio per indirizzo sono costituiti dalla velocità e dal risparmio di spazio per elementi molto estesi.

Il C passa gli array per indirizzo poiché l'inserimento nello stack di un array di 10000 elementi è un'operazione estremamente lenta. Il passaggio di array per valore può essere ottenuto dichiarando gli array come unici membri di una struttura. Il passaggio di tutti gli altri tipi di dati per valore invece che per indirizzo non è un problema in C grazie alla presenza dell'operatore di indirizzamento & che può essere inserito davanti a qualsiasi variabile.

Le dimensioni di un parametro passato per indirizzo (un puntatore) in C dipendono dal modello di memoria impiegato. Con i modelli tiny, small e medium, i puntatori occupano due byte e pertanto viene passato solamente l'offset. Nei modelli di memoria compact, large e huge i puntatori occupano 4 byte e includono il segmento e l'offset. Nel modello flat a 32 bit tutti i puntatori sono offset near a 32 bit.

Tabella 17.1 Convenzioni di chiamata nei linguaggi di alto livello.

	Ordine dei parametri	**Passaggio per valore/indirizzo**	**Reinizializzazione Stack**	**Convenzioni per i nomi**
BASIC	normale	offset	ret #	rimozione delle informazioni relative al tipo (%, &, !, #, $)
C	inverso	valori	chiamante	carattere di sottolineatura iniziale, distinzione maiuscole/minuscole
Fortran	normale	puntatori far	ret #	N/D
Pascal	normale	valori	ret #	N/D

Impostazione dello stack frame

Lo stack frame deve essere impostato quando lo stack viene utilizzato per il passaggio di parametri o quando vi si devono memorizzare variabili locali. Il termine stack frame è utilizzato per descrivere l'istanza temporanea di una struttura di dati sullo stack. Come puntatore allo stack frame viene utilizzato il registro BP (o EBP). Quando si utilizza BP come un puntatore, il segmento dello stack è costituito dal segmento standard. Ecco l'aspetto del codice a 16 bit:

```
push  pb
mov   bp, sp
```

Ecco invece come si può impostare lo stack frame per codice a 32 bit:

```
push  ebp
mov   ebp, esp
```

Allocazione di spazio locale sullo stack

Se è necessario utilizzare una parte dello stack per la memorizzazione delle variabili locali della subroutine assembler, l'operazione deve essere eseguita dopo l'impostazione dello stack frame. Lo spazio viene allocato semplicemente sottraendo il numero di byte richiesti al puntatore allo stack (SP o ESP):

```
sub   esp, 8         ; alloca 8 byte di spazio sullo stack
                     ; per le variabili locali
```

Salvataggio dei registri

I compilatori C si attendono che, all'uscita da una funzione, alcuni registri mantengano immutato il proprio valore. Questi registri sono BP, SI, DI e DS. Per poter eseguire un ritorno corretto al programma chiamante devono essere conservati anche i registri di segmento CS e SS. Le procedure assembler sono dunque libere di modificare AX, BX, CX e DX. Inoltre, se una procedura modifica il flag di direzione (utilizzando CLD o STD) si deve eseguire una PUSH e una POP del registro dei flag.

Caricamento e accesso ai parametri

Dopo aver impostato lo stack frame, dopo aver allocato lo spazio per le variabili locali e dopo aver salvato i registri, si può scrivere il corpo principale della procedura. All'interno di questo codice sarà necessario caricare o accedere ai valori passati attraverso i parametri. La Tabella 17.2 mostra la posizione del primo parametro per ognuno dei modelli di memoria. La posizione del primo parametro rispetto a BP (o EBP) si basa sulle dimensioni dell'indirizzo restituito e dal fatto che BP (o EBP) sia stato salvato nello stack o meno prima dell'impostazione dello stack frame.

Valori restituiti

I valori restituiti dalle funzioni C (scritte in C o in assembler) devono sempre seguire una convenzione. La Tabella 17.3 mostra il modo in cui una funzione C deve restituire vari tipi di dati. Ad esempio, per restituire un intero, si deve inserire il valore nel registro AX.

Tabella 17.2 Posizione del primo parametro sullo stack.

Modello	**Primo parametro**
tiny	[bp+41
small	[bp+4]
compact	[bp+4]
medium	[bp+6]
large	[bp+6]
huge	[bp+6]
flat a 32 bit	[ebp+8]

Tabella 17.3 Convenzioni C riguardanti il valore restituito.

Tipo	**Restituisce il valore in**
(unsigned) character	AL
(unsigned) integer	AX
(unsigned) long integer	DX:AX
Puntatore near	AX
Puntatore far	DX:AX
Float	__fac (Microsoft), FPU ST(0) (Borland)
Double	__fac (Microsoft), FPU ST(0) (Borland)
Long double (10 byte)	FPU st(0)
Strutture near	AX (puntatore alla struttura)
Strutture far	DX:AX (puntatore alla struttura)

Il metodo utilizzato per restituire valori in virgola mobile è abbastanza diverso. Nel caso del C Microsoft, i valori float (precisione semplice) e double (precisione doppia) vengono restituiti nella variabile globale __fac (Floating Point Accumulator). I valori long double vengono restituiti nello stack dell'unità FPU. Nel caso del C Borland, i float, i double e i long double vengono sempre restituiti nello stack dell'unità FPU. Un valore long double è costituito da 10 byte, un formato interno dell'unità FPU. Tutte le funzioni C a 32 bit che utilizzano __fastcall (vedere la discussione presentata in seguito) utilizzano lo stack dell'unità FPU per tutti i tipi di dati in virgola mobile.

Quando una funzione deve restituire una struttura a un programma C, viene restituito un puntatore a tale struttura. Questo significa che una copia della struttura deve

trovarsi in una variabile globale. Nelle convenzioni di chiamata Pascal e Fortran, i compilatori allocano spazio sullo stack del programma chiamante appositamente per la struttura e passano un puntatore a tale area dello stack come un parametro aggiuntivo nel quale dovrà trovarsi il risultato.

Ripristino dello stack per l'uscita

Dopo aver eseguito il codice della funzione, il puntatore allo stack deve assumere nuovamente il valore che aveva prima che fosse allocato lo spazio per lo stack frame e le variabili locali. L'operazione può essere eseguita sommando a SP (o ESP) lo stesso valore che era stato sottratto nel momento in cui era stato allocato lo spazio. Tuttavia, poiché il registro BP (EBP) contiene una copia del registro SP (ESP) prima che fosse allocato lo spazio, è più facile (e talvolta anche più veloce) semplicemente copiare BP (EBP) in SP (ESP). Il registro BP (EBP) deve quindi essere ripristinato dallo stack frame. Infine vi deve essere un'istruzione RET. Per la convenzione di chiamata C, basta utilizzare una semplice RET. Per tutte le altre convenzioni di chiamata, l'istruzione RET deve includere un operando che corrisponde al numero di byte da estrarre dallo stack dopo l'esecuzione dell'istruzione.

```
mov   sp, bp        ; ripristina SP dopo che è stato utilizzato per le variabili locali
pop   bp            ; ripristina BP
ret                 ; convenzione C
```

Ecco il codice di uscita a 32 bit:

```
mov   esp, ebp
pop   ebp
ret
```

Ecco invece il codice di uscita per le convenzioni di chiamata non-C. In questo esempio, la routine ha ricevuto tramite lo stack 4 byte (cioè 2 interi, 2 puntatori near e così via):

```
mov   sp, bp
pop   bp
ret   4
```

Modelli C-assembler

Dopo aver descritto gli aspetti teorici delle chiamate a subroutine assembler, ecco un modello assembler adatto per il modello di memoria small:

(Listato 17.6)

```
push bp             ; salva BP per lo stack frame
mov  bp, sp         ; imposta lo stack frame
sub  sp, 2          ; 2 byte di memoria locale
push di             ; salva i registri
push si

...
mov  ax, [bp+4]     ; accede ai parametri passati
```

```
...
inc  ax
mov  [bp-2], ax       ; usa la memoria locale

...
mov  ax, [bp-2]       ; restituisce un valore intero

...
pop  si               ; ripristina i registri
pop  di
mov  sp, bp           ; dealloca la memoria locale
pop  bp               ; ripristina BP
ret                   ; uscita
```

Per il modello di memoria flat a 32 bit si può utilizzare un modello simile a quello presentato nel Listato 17.7.

(Listato 17.7)

```
push ebp              ; salva EBP per lo stack frame
mov  ebp, esp         ; imposta lo stack frame
sub  esp, 4           ; 4 byte di memoria locale
push  edi             ; salva i registri
push  esi

...
mov  eax, [ebp+8]     ; accede ai parametri passati

...
inc  eax
mov  [ebp-4], eax     ; usa la memoria locale

...
mov  eax, [ebp-4]     ; restituisce un valore intero long

...
pop  esi              ; ripristina i registri
pop  edi
mov  esp, ebp         ; dealloca la memoria locale
pop  ebp              ; ripristina EBP
ret                   ; uscita
```

Scrivendo codice a 32 bit, per le funzioni più semplici (semplici nel senso dell'utilizzo dello stack frame) è possibile utilizzare per lo stack frame ESP invece di EBP. Sia EBP che ESP utilizzano come segmento standard il segmento SS.

(Listato 17.8)

```
push  edi             ; salva i registri
push  esi

...
mov   eax, [esp+12]   ; accede alla prima dword
mov   ebx, [esp+16]   ; accede alla seconda dword

...
```

```
        pop    esi     ; ripristina i registri
        pop    edi
        ret            ; uscita
```

Esempi di chiamata di routine assembler dal C

Dopo aver affrontato il meccanismo di chiamata da un programma C a un modulo assembler, è giunto il momento di provare alcuni esempi. Ecco una versione modificata del Listato 17.4 che consente di utilizzare una funzione assembler esterna.

(Listato 17.9)

```
extern char * strcpy2(char *, const char *);
void main()
{
        char buffer1[80] = "Stringa di esempio";
        char buffer2[80];
        strcpy2(buffer2, buffer1);
        printf("stringa 2 = %s\n", buffer2);
}
```

In un programma C++, la dichiarazione deve essere scritta nel seguente modo:

```
extern "C" char * strcpy2(char *, const char *);
```

L'esempio seguente contiene la stessa funzione strcpy() del Listato 17.5 ma convertita in un file assembler indipendente:

(Listato 17.10)

```
.model small
.code
_strcpy2 proc near

        push   bp
        mov    bp, sp
        push   di
        push   si

        mov    di, [bp+4]      ; string1
        mov    si, [bp+6]      ; string2

lbl:                           ; cicli
        mov    ax, [si]        ; 1
        add    si, 2           ; 0
        cmp    al, 0           ; 1
        je     exit2           ; 0
        mov    [di], ax        ; 1
        add    di, 2           ; 0
        cmp    ah, 0           ; 1
        jne    lbl             ; 0
                               ; —
exit:                          ; 4 cicli, 2 per byte
        jmp    exit3
```

```
exit2:
        mov     [di], al        ; scrive 1 Null
exit3:
        mov     ax, [bp+4]      ; restituisce un puntatore a string1

        pop     si
        pop     di
        pop     bp
        ret

_strcpy2 endp
```

Ecco infine una versione per il modello large della funzione **strcpy()** sempre trasformata in un file distinto:

(Listato 17.11)

```
.model large
.code
_strcpy2 proc far               ; strcpy per il modello large

        push  bp
        mov   bp, sp
        push  di
        push  si
        push  ds
        push  es

        les   di, [bp+6]        ; string1
        lds   si, [bp+10]       ; string2
lbl:
        mov   ax, [si]
        add   si, 2
        cmp   al, 0
        je     exit2
        mov   ES:[di], ax
        add   di, 2
        cmp   ah, 0h
        jne   lbl
exit:
        jmp   exit3
exit2:
        mov   ES:[di], al
exit3:
        mov   ax, [bp+6]        ; restituisce un puntatore a string1
        mov   dx, es

        pop   es
        pop   ds
        pop   si
        pop   di
        pop   bp
        ret
_strcpy2 endp
```

Nella procedura per il modello di memoria large, il primo parametro si trova a 6 byte di distanza da BP (invece di 4) poiché lo stack ha un indirizzo di ritorno far dalla chiamata alla procedura. Il secondo parametro si trova a 4 byte dal primo (invece di 2) poiché nel modello large i puntatori occupano 4 byte invece di 2. Inoltre il codice ha richiesto altre tre modifiche. Innanzitutto si dovevano leggere dallo stack sia i segmenti che gli offset. L'operazione può essere eseguita agevolmente utilizzando le istruzioni LDS e LES. In secondo luogo, poiché vengono caricati puntatori far, è stato necessario inserire (PUSH) ed estrarre (POP) dallo stack i registri di segmento DS ed ES. Infine, ad ogni accesso alla memoria [DI] è stato necessario utilizzare il prefisso di uscita dal segmento ES.

Per questo esempio, la Tabella 17.4 riassume le modifiche richieste per ogni modello di memoria.

Utilizzo della direttiva estesa PROC

Vi sono alcune estensioni della direttiva PROC che rendono l'interfacciamento di codice C e assembler molto più agevole rispetto a quanto è stato possibile vedere dagli esempi precedenti. Nella direttiva PROC è possibile specificare ogni parametro e il relativo tipo di dati e ogni registro che deve essere salvato e ripristinato. Nella direttiva .model, si specifica il modello di memoria e la convenzione di chiamata ad alto livello che si sta utilizzando. Il linguaggio controlla i tre elementi seguenti:

- convenzioni di nome (per i caratteri di sottolineatura in C);
- ordine dei parametri sullo stack;
- note sull'istruzione RET: cancella lo stack o no?

Questo consente di scrivere procedure contenenti meno differenze da un modello di memoria o da un linguaggio a un altro. Il codice presentato nel Listato 17.12 può infatti essere utilizzato con qualsiasi modello di memoria e qualsiasi linguaggio e si occupa di copiare stringhe ASCIIZ.

Tabella 17.4 Riepilogo dei dettagli riguardanti il passaggio di due puntatori in C.

Modello	Posizione di string1	Posizione di string2	Carica puntatori far, salva/ripristina DS/ES ed usa il prefisso ES
tiny	[bp+4]	[bp+6]	no
small	[bp+4]	[bp+6]	no
compact	[bp+4]	[bp+8]	sì
medium	[bp+6]	[bp+8]	no
large	[bp+6]	[bp+10]	sì
huge	[bp+6]	[bp+10]	sì

Nel Listato 17.12 si vede per la prima volta l'impiego dell'assemblaggio condizionale. Questa tecnica consente all'assembler di valutare il valore di un simbolo o di qualche altra espressione per determinare se una determinata sezione di codice debba essere o meno assemblata nel file oggetto. I simboli interni @codesize e @datasize sono definiti dall'assembler sulla base del modello di memoria.

(Listato 17.12)

```
.model small, C ; imposta "small" per qualsiasi modello
.code

strcpy2 PROC USES di si, string1:ptr, string2:ptr

if @datasize
        push    ds
        push    es
        les     di, string1
        lds     si, string2
else
        mov     di, string1
        mov     si, string2
endif

        lbl:
        mov     ax, [si]
        add     si, 2
        cmp     al, 0
        je      exit2
if @datasize
        mov     ES:[di], ax
else
        mov     [di], ax
endif
        add     di, 2
        cmp     ah, 0h
        jne     lbl
        exit:
        jmp     exit3
        exit2:
if @datasize
        mov     ES:[di], al
else
        mov     [di], al
endif
        exit3:
        mov     ax, string1
if @datasize
        mov     dx, es
        pop     es
        pop     ds
endif
        pop     si
        pop     di
```

```
        pop     bp
        ret

strcpy2 endp
```

17.3 Fastcall

Nei capitoli precedenti si è visto come è possibile sfruttare le potenzialità del Pentium per ottimizzare routine assembler; ora si è visto come è possibile interfacciare queste routine con i programmi C e C++. Vi sono anche delle ottimizzazioni che intervengono nell'uso combinato di codice C e assembler. Uno dei motivi per i quali il linguaggio assembler è più veloce e compatto rispetto ai linguaggi di alto livello è il fatto che le procedure assembler vengono normalmente richiamate inserendo i parametri nei registri. Molte volte questi parametri si trovano già nei registri ma se così non fosse, è sempre più veloce spostare i dati in un registro piuttosto che inserirli nello stack. Infatti l'impostazione dello stack frame e il trasferimento dei parametri dallo stack ai registri richiede un gran numero di cicli di CPU.

I linguaggi di alto livello sono realizzati in questo modo poiché il loro scopo è quello di nascondere i dettagli dell'architettura della CPU. Nascondendo il numero e le dimensioni dei registri si è costretti a ricorrere a una soluzione generale, ovvero all'inserimento dei parametri nello stack. Ma questa descrizione non deve trarre in inganno: infatti, l'impiego dello stack per il passaggio dei parametri è una brillante ed elegante innovazione nel campo del software. Tuttavia, molte chiamate a funzioni richiedono solo pochi parametri, che potrebbero senza alcun problema occupare i registri disponibili nell'80x86.

I compilatori C e C++ consentono però di utilizzare la convenzione di chiamata fastcall che permette di inserire nei registri il maggior numero possibile di parametri. Eseguendo la compilazione con l'opzione /Gr si attiva l'opzione fastcall per l'intero file. Utilizzando la parola chiave _fastcall si attiva questa convenzione solo per una determinata funzione. Gli effetti di fastcall sono particolarmente evidenti quando le funzioni richiamate hanno meno di quattro parametri e quando tali funzioni si trovano in un ciclo o vengono richiamate ricorsivamente. Il Borland C/C++ 3.0 e le versioni successive consentono di impiegare la convenzione fastcall tramite la parola riservata _fastcall. A causa di un errore nella convenzione dei nomi del Borland 3.0, è preferibile utilizzare la versione 3.1 o una versione successiva. Il compilatore Microsoft C 6.0 supporta la convenzione fastcall tramite la parola riservata _fastcall. Il Microsoft C/C++ 7.0 e il Visual C++ supportano la convenzione fastcall tramite la parola riservata __fastcall (utilizzata nei seguenti esempi).

Il Listato 17.13 mostra un esempio relativo alla funzione strcpy() tratta dal Listato 17.5.

Misurando questo esempio si scopre, inaspettatamente, che viene eseguito esattamente alla stessa velocità della versione che non impiega la parola riservata __fastcall. Dopo aver osservato il codice assembler generato dal compilatore, si è scoperta una parte di codice molto particolare. Il compilatore infatti inserisce i due parametri nello

stack all'inizio della funzione. Il manuale Microsoft dice di "non utilizzare la convenzione di chiamata __fastcall per le funzioni contenenti blocchi __asm". Inoltre dice che i registri assegnati a un determinato parametro potrebbero cambiare nelle future versioni del compilatore. La domanda, dunque, è: allora __fastcall presenta o non presenta vantaggi rispetto alla codifica in assembler? Ciò che accade è che il compilatore rileva il blocco __asm all'interno della funzione e quindi decide di inserire i parametri nello stack. Il vantaggio rimane però per le funzioni scritte interamente in C.

Per poter ottenere aumenti di prestazioni dal linguaggio assembler e dalla convenzione fastcall, è necessario scrivere la funzione in linguaggio assembler come un modulo distinto. Il modulo distinto può essere scritto in C o assembler. Se si scrive il codice in assembler, si noti che la convenzione C che prevede l'aggiunta di un carattere iniziale di sottolineatura è cambiata e viene richiesto l'utilizzo di un segno @ iniziale. Inoltre, la convenzione di chiamata per i parametri che non vengono passati tramite i registri è cambiata: in pratica si utilizza la convenzione di chiamata Pascal.

(Listato 17.13)

```
char * __fastcall strcpy2(char *string1, const char *string2)
{
 __asm{
        mov   di, string1
        mov   si, string2
        lbl:
        mov   ax, [si]
        add   si, 2
        cmp   al, 0h
        je    exit2
        mov   [di], ax
        add   di, 2
        cmp   ah, 0h
        jne   lbl
        exit:
        jmp   exit3
        exit2:
        mov   [di], al
        exit3:
        mov   ax, string1       ; restituisce il puntatore
  }
}
```

ATTENZIONE *Se si vogliono sfruttare i vantaggi offerti dalla convenzione di chiamata* fastcall *con il codice in linguaggio assembler, sarà necessario eseguire il test di ogni funzione ogni volta che si aggiorna la versione del compilatore.*

Fastcall: gestione dei registri

Per utilizzare in modo efficace la convenzione fastcall, si deve conoscere quali parametri vengono passati in ogni registro. Nella Tabella 17.5 viene presentato un riepilogo dei registri utilizzati dai compilatori Borland e Microsoft per il codice a 16 bit. Tali

convenzioni sono diverse quando si scrive codice assembler a 32 bit con **fastcall**; a tale proposito si consulti la documentazione del compilatore.

Prestazioni del codice C

È facile utilizzare la libreria Timer per misurare le prestazioni del codice C o C++. Occorre però assicurarsi di misurare codice con una durata sufficiente o i risultati ottenuti non saranno sufficientemente precisi. Il Listato 17.14 mostra un esempio di valutazione della funzione **strcpy()** della libreria C:

```
void main()                                                    (Listato 17.14)
{
 int i;
 char buffer1[20] = "Stringa di esempio ";
 char buffer2[20];

 timer_on();                 /* avvia il timer          */
 for(i=0;i<1000;i++)
 {
        strcpy(buffer2, buffer1);
 }
 timer_off();                /* ferma il timer          */
 timer_show();               /* visualizza i risultati  */
}
```

La Tabella 17.6 mostra i risultati delle funzioni **strcpy()** sviluppate nel Capitolo 13 e utilizzate in varie configurazioni anche in questo capitolo.

Tabella 17.5 Gestione dei registri in fastcall (codice a 16 bit).

Tipi	Registri utilizzabili
char	AL, DL, BL
unsigned char	AL, DL, BL
int	AX, DX, BX
unsigned int	AX, DX, BX
long int	DX:AX
unsigned long int	DX:AX
puntatore near	BX, AX, DX (Microsoft)
puntatore near	AX, DX, BX (Borland)
puntatore far	solo sullo stack
float, double e altri	solo sullo stack

Tabella 17.6 Prestazioni della funzione strcpy() per il modello small misurata in microsecondi (55 caratteri copiati 1000 volte).

	Pentium-60	**486-33**	**386-25**
strcpy() della libreria C	6002	13350	32950
strcpy() inline assembler	2367	9716	38300
strcpy() fastcall	2251	9343	34560

Dalla Tabella 17.6 si può osservare che la routine assembler velocizza la copia di stringhe del 37% su un 486 e del 150% su un Pentium. Quando viene impiegata la convenzione fastcall, l'accelerazione è perfino maggiore: 42% su un 486 e 166% su un Pentium. Le misurazioni presentate nella Tabella 17.6 si basano su una stringa lunga 55 caratteri. Come si può vedere nella Figura 17.1 questa lunghezza può essere considerata significativa.

Dal grafico presentato nella Figura 17.1 si può vedere che nel caso delle stringhe più brevi, l'accelerazione è di due o tre volte. Tale accelerazione cala lentamente a 100 o 200 caratteri.

Al contrario su un 386 le prestazioni della funzione strcpy() realizzata in assembler sono peggiori (dal 5% al 14% per stringhe di 55 caratteri). Le prestazioni del 386 sono invece migliori nel caso delle stringhe più piccole e circa uguali per stringhe lunghe 10-35 caratteri.

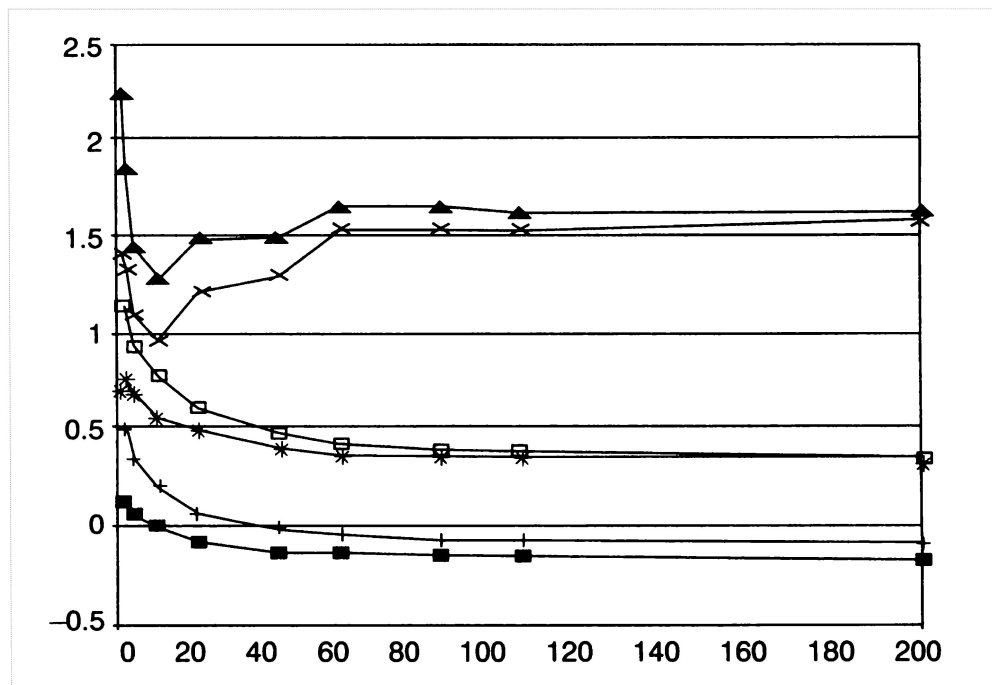



Figura 17.1 Rapporto fra lunghezza della stringa e velocità.

Sono molti i casi in cui è possibile migliorare le prestazioni dei programmi C e C++ utilizzando routine in linguaggio assembler. Questo fatto è particolarmente vero nel caso del 486 e del Pentium. Se si ricercano le massime prestazioni, non ci si deve illudere che tutto il lavoro di ottimizzazione possa essere eseguito dal compilatore. In particolare si devono considerare e isolare i punti del codice C o C++ che richiedono una migliore ottimizzazione e passare dalle false promesse ai fatti.

Capitolo 18

Programmazione in modalità protetta

In questo capitolo si parlerà della programmazione in modalità protetta e della realizzazione di software "invisibile".

18.1 Introduzione alla modalità protetta

Sono molti gli ambienti che consentono di eseguire programmi in modalità protetta e fra questi vi sono Windows, Windows NT, OS/2 e i programmi di espansione del DOS. Possono essere utilizzati anche altri sistemi operativi come lo UNIX SCO ed altre versioni. Le piattaforme di gran lunga più diffuse sono DOS e Windows e dunque ci si limiterà alla descrizione della modalità protetta in questi ambienti. Il DOS funziona in modalità reale (sempre a 16 bit) o in modalità 8086 virtuale (sempre a 16 bit) sotto il controllo di un gestore di memoria. Windows 3.0 o 3.1, se eseguiti in modalità avanzata, funzionano in modalità protetta a 16 bit. Windows 95 e Windows NT funzionano in modalità protetta a 32 bit.

La differenza fra la modalità protetta a 16 bit e la modalità protetta a 32 bit è il fatto che i sistemi operativi a 32 bit devono riconoscere e consentire l'esecuzione di programmi contenenti codice a 32 bit. Tali programmi contengono segmenti dichiarati come USE32. Questo tipo di segmenti, prima dell'inizio dell'esecuzione deve essere impostato in modo opportuno dal sistema operativo.

I descrittori di segmento dal 386 in poi contengono un bit chiamato bit D. Il bit D, in un segmento di codice, indica la lunghezza standard per gli operanti e gli indirizzi effettivi. Quando il bit D è uguale a 1, si utilizzano operandi e indirizzi effettivi a 32 bit. Quando il bit D è uguale a 0, si impiegano operandi e modalità di indirizzamento a 16 bit. Le dimensioni degli operanti e/o degli indirizzi effettivi possono anche essere modificate istruzione per istruzione impiegando il prefisso per gli operandi (66h) e/o il prefisso dimensionale degli indirizzi (67h). Questi registri vengono inseriti automaticamente dall'assembler sulla base degli operandi di ciascuna istruzione e senza che sia necessario specificare un segmento come USE32.

Le estensioni DOS, come ad esempio Phar Lap TNT, consentono ai programmi DOS di funzionare in modalità protetta traducendo le chiamate al DOS e al BIOS in chiamate a 16 bit in modalità reale. Alcune espansioni DOS consentono solo la realizzazione di codice in modalità protetta operante a 16 bit; altre consentono l'impiego di codice in modalità protetta a 16 e 32 bit. I produttori di queste estensioni DOS in genere forniscono particolari toolkit che richiedono il pagamento dei relativi diritti per le librerie run-time utilizzate per l'esecuzione dei programmi creati.

18.2 L'interfaccia DPMI (DOS Protected-Mode Interface)

L'ambiente di sviluppo scelto si basa spesso sulle preferenze dell'utente, sulle dimensioni del mercato, sulla base installata e così via. Questi argomenti non sono rilevanti per questa discussione poiché si parlerà di un ambiente ampiamente disponibile e che costituisce un'eccellente piattaforma di test per altri ambienti. Questa piattaforma è l'interfaccia DPMI o DOS Protected-Mode Interface. L'interfaccia DPMI (brevemente descritta nel Capitolo 6) è costituita da specifiche che consentono ai programmi DOS di accedere alle funzionalità avanzate dei microprocessori 80286 (e successivi) utilizzando un approccio standard.

Anche se le applicazioni che utilizzano direttamente l'interfaccia DPMI possono avere un valore commerciale limitato, l'utilizzo di programmi DOS DPMI per valutare le prestazioni del codice, specialmente per il Pentium, può essere un'opportunità molto interessante. Infatti, quando si utilizza un ambiente multitasking come Windows o OS/2, è difficile conservare il controllo completo della CPU e questo fattore è fondamentale per eseguire valutazioni accurate delle prestazioni dei programmi.

Si supponga di voler scrivere routine come quelle operanti sulle stringhe e descritte nel Capitolo 14. Non vi sono problemi se si scrive codice a 16 bit poiché è possibile scrivere ed eseguire il test del codice in DOS. Ma cosa accade se si vuole scrivere codice a 32 bit? Si può scrivere ed eseguire il test del codice in DOS ma non sarà possibile valutarne le prestazioni. Assemblando il codice per il DOS, l'assembler crea codice a 16 bit per segmenti a 16 bit. Questo significa che tutte le operazioni a 32 bit

dovranno essere precedute dai prefissi di modifica delle dimensioni dell'operando o dell'indirizzo. Si ricordi la Regola 6 tratta dalla Tabella 9.2:

- Le istruzioni con prefisso possono essere eseguite solo nella pipe U (ad eccezione del prefisso 0F di Jcc).

Ad esempio:

```
mov   eax, 1
mov   ebx, 2
```

Queste istruzioni sono accoppiabili nel codice a 32 bit e la loro esecuzione richiede un solo ciclo. Ma se il codice è a 16 bit, queste istruzioni non sono accoppiabili e il prefisso richiede un ciclo aggiuntivo per ogni istruzione. Il risultato finale produce due istruzioni che richiedono due cicli ciascuna, ovvero codice quattro volte più lento a 16 bit rispetto allo stesso codice in modalità a 32 bit.

Utilizzando un server DPMI è possibile scrivere, verificare con precisione le prestazioni ed eseguire codice a 32 bit. Gli strumenti necessari sono:

- un assembler e un linker che generino codice a 32 bit;
- un gestore di memoria comprendente un server DPMI.

Gli assembler TASM 1.0 (o successivi) e MASM 5.1 (o successivi - preferibilmente a partire dal MASM 6.0) generano entrambi codice a 32 bit e i rispettivi linker (TLINK e LINK) producono file eseguibili a 32 bit. Come server DPMI può essere utilizzato il gestore di memoria 386MAX versione 6.01 o 7.0.

18.3 Segmenti in modalità protetta

In modalità protetta (sia a 16 che a 32 bit), il valore contenuto nei registri di segmento è chiamato selettore. Si tratta di un puntatore a una tabella dei descrittori che contiene informazioni relative al segmento, ad esempio l'indirizzo iniziale, la lunghezza totale, il bit D (di cui si è parlato in precedenza) e così via. Tutte le informazioni contenute nella tabella dei descrittori vengono copiate nei registri interni nel momento in cui viene caricato un registro di segmento. Questo è il motivo per cui il caricamento di registri di segmento in modalità protetta è più lento rispetto all'analoga operazione in modalità reale. Ogni volta che si calcola un indirizzo in modalità reale, devono essere combinati fra loro il segmento e l'offset e il segmento deve essere fatto scorrere verso sinistra di 4 bit. In modalità protetta l'offset viene sommato all'indirizzo iniziale del segmento.

18.4 Conversione di codice in modalità protetta

Per la differenza esistente fra segmenti e selettori, in genere non vi sono operazioni che possono essere eseguite sui selettori. La conversione di codice dalla modalità rea-

le alla modalità protetta richiede quindi l'eliminazione di tutte le sequenze di codice contenenti operazioni aritmetiche sui segmenti.

Ma per trasformare il codice per la modalità protetta sono necessarie anche altre modifiche:

- non è possibile scrivere dati sul segmento di codice;
- alcune istruzioni, ad esempio CLI, STI, IN e OUT, possono provocare violazioni della protezione.

18.5 Programmazione in modalità protetta mista a 16 e 32 bit

Questo tipo di utilizzo misto di codice a 16 e 32 bit nello stesso programma è molto impiegato dal software per sistemi operativi e per utility oltre al test di codice a 32 bit in un ambiente operativo a 16 bit, così come si proverà a fare in questo capitolo. Il motivo che spinge a unire codice a 16 e 32 bit in un'applicazione è principalmente quello di consentire l'esecuzione di codice a 32 bit in un ambiente a 16 bit. Dunque un'applicazione a 16 bit potrebbe, se necessario, eseguire codice a 32 bit, sempre che ciò sia consentito dall'ambiente operativo impiegato.
Talvolta, per descrivere le varie combinazioni di segmenti, selettori e offset viene impiegata la seguente terminologia:

16:16	segmento:offset	modalità reale
16:16	selettore:offset	modalità protetta a 16 bit
16:32	segmento: offset	modalità virtuale 8086 con indirizzi anche a 32 bit
16:32	selettore:offset	modalità protetta a 32 bit
0:32	offset	modello flat a 32 bit

18.6 Definizioni complete dei segmenti

Le definizioni complete dei segmenti sono necessarie quando si deve avere un controllo completo sui segmenti. È comunque possibile utilizzare insieme combinazioni di segmenti semplificati e segmenti completi ma questi non devono entrare in conflitto con le definizioni automatiche dei segmenti utilizzate dall'assembler. Le direttive di segmentazione semplificata consentono di creare segmenti USE32 inserendo la direttiva .386, .486 o .586 prima della direttiva .model (la direttiva .586 consente di utilizzare le nuove istruzioni del Pentium). La direttiva .model consente di impostare come standard la modalità USE32 dalla quale è possibile uscire temporaneamente utilizzando la direttiva USE16.

La Tabella 18.1 descrive la sintassi completa della direttiva SEGMENT nelle ultime versioni di MASM (versione 6.0 e successive) e TASM (versione 3.1 e successive).

Tabella 18.1 Definizione di segmenti con la direttiva SEGMENT.

segmento SEGMENT [*allineamento*] [READONLY] [*combina*] [*use*] ['*classe*']
    codice e/o dati
segmento ENDS

dove:	
[]	Gli elementi fra parentesi quadri sono connazionali.
segmento	Il nome scelto per il segmento.
SEGMENT	Il nome della direttiva.
allineamento	Il tipo di allineamento (a byte, word, dword, paragrafo o pagina - l'impostazione standard è a paragrafo).
READONLY	Parola riservata che dichiara che il segmento è di sola lettura. Per ogni istruzione che modifica un oggetto contenuto in questo segmento, l'assembler genera un messaggio d'errore.
combina	Determina il modo in cui il linker riunisce i segmenti. Può essere: PRIVATE, PUBLIC, STACK, COMMON, MEMORY o AT *indirizzo*.
use	USE16 o USE32.
classe	Specifica un nome di classe per il segmento. Il linker raggruppa insieme i segmenti con lo stesso nome di classe.
ENDS	Direttiva richiesta per concludere un segmento.
Tutti i parametri opzionali che seguono la parola riservata SEGMENT possono essere specificati in qualsiasi ordine.	
Parametro *combina*:	
PRIVATE (standard)	Il segmento non viene combinato con altri aventi lo stesso nome ma in altri moduli.
PUBLIC	Tutti i segmenti aventi lo stesso nome ma in moduli distinti vengono combinati in un unico segmento contiguo. Normalmente è quello che si desidera.
STACK	Necessario per dati non inizializzati per lo stack.
COMMON	Segmenti che si sovrappongono. La lunghezza finale corrisponde al massimo dei segmenti combinati. Non adatto a dati inizializzati.
MEMORY	Come PUBLIC.
AT *indirizzo*	Posiziona il segmento all'indirizzo specificato. Solo per la modalità reale. Il link non viene eseguito dal linker; utilizzato solo per definire i dati o le strutture a determinati indirizzi far.
Parametro *use*:	
USE16	Situazione standard: offset e operandi a 16 bit.
USE32	Situazione standard: offset e operandi a 32 bit.
FLAT	
Se prima di una direttiva .model viene specificata .386, .486 o .586, il tipo standard è USE32, altrimenti è USE16.	

18.7 Valutazione di programmi in modalità protetta

La versione a 32 bit della libreria Timer (vedere il Capitolo 12) consente di valutare le prestazioni del codice in modalità protetta. Questa libreria (TIMER32.LIB) contiene le dichiarazioni di segmenti corrette per il codice a 32 bit. In modalità protetta, le istruzioni CLI (CLear Interrupt flag) e STI (SeT Interrupt flag) normalmente provocano un errore di protezione generale. Per questo motivo, non è facile disabilitare gli interrupt. Dunque è importante valutare più eventi e tenere in considerazione che nel frattempo possono essersi verificati degli interrupt.

18.8 Modello di codice a 32 bit per la modalità protetta

Il codice presentato nel Listato 18.1 è un modello utilizzabile per realizzare codice a 16 e/o 32 bit operante in modalità protetta.

(Listato 18.1)

```
.model small
.stack
.486

.data

intro_msg           db 'Ingresso in modalità protetta.',13,10,0
in16_msg            db 'In modalità protetta, segmento a 16 bit.',13,10,0
in32_msg            db 'In modalità protetta, segmento a 32 bit.',13,10,0

; dati restituiti da INT 2Fh funzione 1687h

DPMI_flags          db 0
CPU_type            db 0
DPMI_major_version  db 0
DPMI_minor_revision db 0
DPMI_mem            dw 0
DPMI_entry          dd 0

code32_selectordw 0                   ; contiene il selettore del segmento CODE32
main32_address      df main32

cs_descriptor label dword
cs_limitdw 0                          ; limite del descrittore
cs_low_addr         db 0,0,0          ; 3 byte inferiori dell'indirizzo
cs_access           dw 0              ; descrittore dei byte di accesso
cs_high_addr        db 0              ; byte superiore dell'indirizzo

; messaggi d'errore

err_msg1            db 'Host DPMI non rilevato',13,10,0
err_msg2            db 'Impossibile allocare la memoria richiesta in modalità reale',13,10,0
err_msg3            db 'La commutazione in modalità protetta non ha funzionato',13,10,0
```

```
.code

main proc

        mov     ax, @data
        mov     ds, ax

; (1)   ridimensiona la memoria allocata

; ↑ ——— note descritte nel manuale

        mov     bx, ss              ; inizio del segmento dello stack
        mov     ax, es              ; inizio PSP
        sub     bx, ax              ; paragrafo per codice e dati
        mov     ax, sp              ; determina le dimensioni dello stack
        shr     ax, 4               ; divide per 16 per conoscere i paragrafi
        add     bx, ax
        inc     bx                  ; BX = paragrafi richiesti
        mov     ah, 4ah             ; modifica l'allocazione della memoria
        int     21h

; (2)   controlla se sono disponibili i servizi DPMI

        call    DPMI_check          ; verifica la presenza di un server DPMI
        jc      error1              ; esce se non c'è

; (3)   Stampa del messaggio introduttivo

        lea     si, intro_msg       ; messaggio introduttivo
        call    print_string

; (4)   Alloca la memoria per il server DPMI

        mov     ax, 0               ; usa seg=0 se non è richiesta memoria
        mov     bx, DPMI_mem        ; determina il numero di paragrafi richiesti dal DPMI
        test    bx, bx
        jz      main_2              ; salta se non è richiesto
        mov     ah, 48h             ; alloca la memoria
        int     21h
        jc      error2              ; verifica di errori di allocazione

; (5)   commutazione in modalità protetta
main_2:
        mov     es, ax
        mov     ax, 1               ; indica applicazione a 32 bit
        call    dword ptr DPMI_entry ; entra in modalità protetta
        jc      error3              ; se CF=1, errore di commutazione

; (6)   stampa il messaggio di ingresso in modalità protetta

        lea     si, in16_msg        ; messaggio di ingresso in modalità protetta a 16 bit
        call    print_string
```

```
; <<< richiamare il codice in modalità protetta a 16 bit >>>

;=======================================
; inizio impostazione della modalità protetta a 32 bit
;=======================================

; (7)   alloca un descrittore per il segmento use32

        mov     ax, 0                           ; alloca il descrittore LDT
        mov     cx, 1                           ; 1 descrittore
        int     31h                             ; richiama l'host DPMI
        mov     code32_selector, ax             ; selettore del segmento use32
        mov     word ptr main32_address+2, ax   ; memorizza il selettore

; (8)   ottiene una copia dell'elemento LDT

        mov     bx, cs                          ; ottiene il descrittore CS corrente
        mov     di, offset cs_descriptor        ; punta al buffer risultante
        movzx   edi, di                         ; (converte a 32 bit)
        push    ds                              ; ES:EDI => buffer
        pop     es                              ;
        mov     ax, 000bh                       ; ottiene la richiesta del descrittore
        int     31h                             ; richiama l'host DPMI

; (9)   imposta l'indirizzo di base del segmento

        mov     ax, seg code32                  ; indirizzo lineare di code32
        movzx   eax, ax                         ;
        shl     eax, 4                          ; valore del segmento per 16
        mov     dx, ax                          ; spostamento in cx:dx
        mov     ecx, eax                        ;
        shr     ecx, 16                         ;
        mov     bx, code32_selector             ; identifica il selettore
        mov     ax, 7                           ; imposta l'indirizzo di base del segmento
        int     31h                             ; richiama l'host DPMI

; (10)  imposta il limite del segmento

        mov     cx, 0000h                       ; imposta il limite del segmento a 64K
        mov     dx, 0ffffh                      ;
        mov     bx, code32_selector             ; identifica selettore
        mov     ax, 8h                          ; imposta il limite del segmento
        int     31h                             ; richiama l'host DPMI

; (11)  entra in modalità a 32 bit modificando il bit D

        mov     cx, cs_access                   ; inizia con l'indirizzo corrente di cs
        or      ch, 40h                         ; imposta il valore standard a 32 bit
        mov     bx, code32_selector             ; identifica il selettore
        mov     ax, 0009h                       ; imposta la richiesta dei diritti di accesso
        int     31h                             ; richiama l'host DPMI
```

```
; (12)  Esegue una chiamata far 16:16 al codice a 32 bit

        call    dword ptr main32_address

; All'uscita si è ancora in modalità protetta a 16 bit.

main_exit:
        mov     ax, 4c00h               ; Uscita al DOS
        int     21h

; messaggi d'errore

error1:
        lea     si, err_msg1            ; host DPMI
        jmp     err0
error2:
        lea     si, err_msg2            ; allocazione della memoria
        jmp     err0
error3:
        lea     si, err_msg3            ; commutazione in modalità protetta
err0:
        call    print_string
        jmp     main_exit

main endp


DPMI_check proc near
;-----------------------------------------------
; Verifica la presenza del server DPMI
;
; output: CF=1 se non è installato un server DPMI
;-----------------------------------------------

        push    ax
        push    bx
        push    cx
        push    dx
        push    di
        push    si
        push    es

        mov     ax, 1687h               ; richiede l'indirizzo dell'host DPMI
        int     2fh                     ; tramite un interrupt multiplex (2Fh)
        test    ax, ax                  ; AX = 0 = successo
        jnz     not_found               ; altrimenti esce con un errore

        and     bl, 00000001b           ; verifica il bit 1 per vedere se è supportato
        setne   al                      ; il codice a 32 bit
        mov     DPMI_flags, al
```

```
        mov     CPU_type, cl
        mov     DPMI_major_version, dh
        mov     DPMI_minor_revision, dl
        mov     DPMI_mem, si
        mov     word ptr DPMI_entry[0], di
        mov     word ptr DPMI_entry[2], es
        clc                                     ; successo
dc_ret:
        pop     es
        pop     si
        pop     di
        pop     dx
        pop     cx
        pop     bx
        pop     ax
        ret

not_found:
        stc
        jmp     dc_ret

DPMI_check  endp

public print_string
print_string proc near
;------------------------------------------
; Stampa una stringa ASCIIZ
;
; input: DS:SI ptr alla stringa
;------------------------------------------

        push    ax
        push    bx
        push    cx
        push    dx

        call    str_len
        mov     dx, si
        mov     bx, 1                           ; stdout
        mov     ah, 40h                         ; handle di scrittura del DOS
        int     21h

        pop     dx
        pop     cx
        pop     bx
        pop     ax
        ret

print_string endp

public str_len
```

```
str_len proc near
;------------------------------------------------
; Determina la lunghezza della stringa
;
; input: DS:SI ptr alla stringa
;
; output: CX lunghezza
;------------------------------------------------

        push    ax
        push    si

        lea     cx, [si+2]
sl_1:
        mov     ax, [si]
        add     si, 2
        test    al, al
        je      sl_2
        test    ah, ah
        jne     sl_1
        inc     si
sl_2:
        sub     si, cx
        mov     cx, si

        pop     si
        pop     ax
        ret

str_len endp

code32  segment public para use32 'code'
        assume  cs:code32

public main32
main32 proc far
;------------------------------------------------
; punto d'ingresso del codice a 32 bit
;
; DS = @data dal codice a 16 bit
;------------------------------------------------

        pusha

; (13)  visualizza il messaggio di ingresso in modalità a 32 bit

        mov     esi, offset in32_msg
        call    print_string32

; (14)  richiama il codice a 32 bit
```

```
        call    do_test

; ripristina i registri

        popa

; (15)  restituisce il segmento di codice a 16 bit da cui si proviene

        db      66h             ; prefisso per le dimensioni dell'operando (vedere nota)
        ret

;--------------------------------------------------------------------------------
; Nota: se fosse stata utilizzata una RET senza il prefisso dimensionale
;       dell'operando, l'assembler avrebbe generato una comune RETF per codice
;       a 32 bit (16:32). Questo è necessario per rientrare alla CALL che ha condotto qui (16:16).
;--------------------------------------------------------------------------------

main32 endp

do_test proc

        ret                     ; questo esempio non fa nulla

do_test endp

public print_string32
print_string32 proc near
;-----------------------------------------
; Stampa a 32 la stringa ASCIIZ
;
; input: DS:ESI ptr alla stringa
;-----------------------------------------

        push    eax
        push    ebx
        push    ecx
        push    edx

        call    str_len32
        mov     edx, esi
        mov     ebx, 1  ; stdout
        mov     ah, 40h ; handle di scrittura DOS
        int     21h

        pop     edx
        pop     ecx
        pop     ebx
        pop     eax
        ret

print_string32 endp
```

```
public str_len32
str_len32 proc near
;--------------------------------------------------
; Determina a 32 la lunghezza della stringa
;
; input: DS:ESI ptr alla stringa
;
; output: ECX lunghezza
;--------------------------------------------------

        push    eax
        push    esi

        lea     ecx, [esi+2]
sl32_1:
        mov     eax, [esi]
        add     esi, 2
        test    al, al
        je      sl32_2
        test    ah, ah
        jne     sl32_1
        inc     esi
sl32_2:
        sub     esi, ecx
        mov     ecx, esi

        pop     esi
        pop     eax
        ret

str_len32 endp

code32  ends

end main
```

NOTA *Le funzioni DPMI utilizzate nel Listato 18.1 sono documentate in* "DPMI Specification version 1.0" *disponibile presso Intel, numero di ordine 240977-001.*

Ecco una breve descrizione del codice contenuto nel Listato 18.1. La numerazione corrisponde ai numeri presenti nei commenti del codice.

1. Il programma inizia ridimensionando la memoria alla quantità effettivamente utilizzata dal programma. Se si cambia la configurazione dei segmenti, occorre assicurarsi di modificare anche questo valore. Il calcolo si basa sullo stack alla fine dell'immagine di memoria.

2. Verifica la presenza di un server DPMI. Se non esiste, il programma termina con un messaggio d'errore.

3. Viene visualizzato il messaggio iniziale.

4. Viene allocata, se necessario, la memoria per il server DPMI. Quando viene eseguita la verifica della presenza di un server DPMI, la chiamata restituisce il numero di paragrafi di memoria necessari. Tale valore può essere uguale a 0.

5. Passa in modalità protetta. Quando viene eseguita la verifica della presenza di un server DPMI, viene restituito l'indirizzo far della procedura utilizzata per entrare in modalità protetta. Questo punto d'ingresso viene utilizzato solo per la prima commutazione in modalità protetta eseguita dall'applicazione.

   Quando il passaggio in modalità protetta sarà stato completato, l'applicazione acquisisce il controllo con i seguenti risultati:

   CS = all'ingresso il selettore è CS con dimensioni di 64 KB
   DS = all'ingresso il selettore è DS con dimensioni di 64 KB
   SS = all'ingresso il selettore è SS con dimensioni di 64 KB
   ES = selettore per PSP (dimensioni = 100h)
   FS = 0
   GS = 0

   Se all'ingresso SS e DS risultano uguali, verrà allocato un solo descrittore e all'uscita sia SS che DS avranno lo stesso valore. Il puntatore d'ambiente (in PSP:2Ch) in PSP viene automaticamente convertito in un selettore valido (solo nel caso in cui all'ingresso non sia uguale a zero).

6. Viene visualizzato un messaggio che conferma l'avvenuto ingresso in modalità protetta.

7. Viene allocato un descrittore per il segmento USE32. Anche se il codice a 32 bit si trova in memoria, il server DPMI non sa nulla di tale codice. Si deve dunque allocare un descrittore per questo segmento.

8. Dopo l'allocazione del descrittore, si deve ottenere una copia del punto di ingresso nella tabella LDT (Local Descriptor Table) in modo da potervi apportare modifiche.

9. Imposta l'indirizzo di base del segmento. La prima modifica consiste nello specificare l'indirizzo iniziale del segmento di codice a 32 bit.

10. Imposta i limiti del segmento. La seconda modifica consiste nello specificare le dimensioni del segmento di codice a 32 bit. Poiché tale codice è stato caricato dal DOS in memoria bassa, le sue dimensioni sono limitate a 640 KB proprio come ogni altro programma DOS. Per eseguire un programma di maggiori dimensioni si deve allocare la memoria sotto 1 MB e caricare il programma nella memoria allocata.

11. L'ultima modifica consiste nell'identificazione del segmento di codice contenente il programma come un segmento di codice a 32 bit modificando il bit D nei byte contenenti i diritti di accesso e il tipo. Quando un server DPMI alloca un descrittore, il valore standard del bit D è 0, ovvero il segmento di codice è a 16 bit.

12. Il passo successivo consiste nell'esecuzione di una chiamata al codice a 32 bit. Si tratta di una chiamata far in modalità protetta 16:16. Questo significa che vi è un indirizzo di memoria costituito da un offset a 16 bit seguito da un selettore a 16 bit.

Questo indirizzo viene memorizzato nel segmento dati in una variabile chiamata main32_address. Il tipo di main32_address deve sempre essere dword (DD), anche se si cambia modello di memoria. Varie versioni di assembler possono generare errori e richiedere una dichiarazione DF. Questo non è un problema poiché main32_address è dichiarata come puntatore a main32 ma la seconda word viene sempre sovrascritta dall'effettivo selettore per fornire sempre un puntatore dword corretto per il codice a 16 bit.

13. Il codice a 32 bit visualizza un messaggio. È importante notare che le routine utilizzate sono diverse dalle utility per codice a 16 bit.
14. Questo è il punto dove si deve inserire una chiamata al codice a 32 bit da valutare.
15. Questo è il rientro al segmento di codice a 16 bit da cui si proviene. Questa operazione può sembrare strana ma è molto importante. Infatti questo rientro deve corrispondere alla chiamata. Un rientro near per un segmento USE32 è un offset a 32 bit. Un rientro far per un segmento USE32 è un puntatore far costituito da 6 byte (32 bit per l'offset e 16 bit per il selettore). La chiamata alla procedura main32 è una chiamata far a 32 bit (16 bit per l'offset e 16 bit per il selettore). Per eseguire l'istruzione RET corretta è necessario utilizzare il byte di prefisso per la definizione delle dimensioni dell'operando (66h). Nella maggior parte dei casi, gli assembler MASM e TASM sono in grado di generare automaticamente i dati di prefisso. In questo caso non vi è altra scelta se non inserire manualmente il byte di prefisso. Questa istruzione con prefisso è necessaria solo se si deve tornare al codice originale a 16 bit in modalità protetta. In caso contrario si può semplicemente eseguire un'istruzione INT 21h, funzione 4Ch e uscire al DOS.

Segmenti dati di grandi dimensioni

Così come è scritto, il programma modello ha accesso solo a codice a 32 bit caricato dal DOS nei primi 640 KB di memoria. Inoltre, questo esempio utilizza solo un segmento dati di dimensioni pari a 64 KB (o meno) accessibile da codice per la modalità reale o virtuale 8086, per la modalità protetta a 16 bit e per la modalità protetta a 32 bit. Per utilizzare un segmento dati di maggiori dimensioni si possono impiegare due tecniche.

Innanzitutto, i dati possono essere dichiarati nel file sorgente in un segmento dichiarato USE32 con la lunghezza desiderata. Questo metodo ha un limite poiché i dati devono rientrare nei primi 640 KB al momento in cui il DOS carica il programma (per utilizzare insieme segmenti USE16 e USE32 in un programma con segmenti più grandi di 64 KB sembra sia necessario che non vengano utilizzati segmenti semplificati).

Il secondo metodo consiste nell'allocare un blocco di memoria estesa tramite i servizi DPMI. Non si tratta di un sistema semplice come l'allocazione della memoria da DOS. La memoria deve essere allocata con una chiamata a funzione e il selettore viene ottenuto tramite un'altra chiamata a funzione in modo da allocare un descrittore. Infine devono essere impostati l'indirizzo di base del descrittori e le dimensioni del blocco di memoria.

```
; Alloca un segmento dati esteso (Listato 18.2)
;
; input:  BX:CX  dimensioni del blocco (la granularità deve essere in blocchi di 4K)
;         DS     segmento dati
;
; output: CF = 1 = errore
;-----------------------------------------------------------

        mov     temp_size, cx
        mov     temp_size[2], bx

; alloca la memoria da DPMI

        mov     ax, 0501h               ; alloca la memoria
        int     31h
        jc      lall_ret

        mov     temp_base, cx
        mov     temp_base[2], bx
        mov     temp_handle, di
        mov     temp_handle[2], si

; alloca un descrittore per il segmento use32

        mov     ax, 0                   ; alloca il descrittore LDT
        mov     cx, 1                   ; 1 descrittore
        int     31h                     ; richiama l'host DPMI
        jc      lall_ret
        mov     temp_selector, ax       ; salvataggio temporaneo

; imposta l'indirizzo di base del segmento

        mov     bx, ax                  ; selettore
        mov     dx, temp_base
        mov     cx, temp_base[2]
        mov     ax, 7                   ; imposta l'indirizzo di base del segmento
        int     31h                     ; richiama l'host DPMI

; imposta il limite del segmento

        mov     dx, temp_size
        mov     cx, temp_size[2]
        mov     bx, temp_selector       ; selettore
        mov     ax, 8h                  ; imposta il limite del segmento
        int     31h                     ; richiama l'host DPMI

        mov     ax, temp_selector
        mov     di, temp_handle
        mov     si, temp_handle[2]

lall_ret:
```

```
        ret

large_allocate endp

large_deallocate proc

        ; input: SI:DI  handle del blocco di memoria
        ;        BX selettore


        mov     ax, 1           ; libera il descrittore LDT
        int     31h             ; DPMI

        mov     ax, 0502h       ; libera il blocco di memoria
        int     31h             ; DPMI
        ret

large_deallocate endp
```

Per utilizzare la procedura large_allocate presentata nel Listato 18.2, si deve specificare nella coppia di registri BX:CX la memoria da allocare. Poiché la memoria viene allocata in pagine da 4 KB, le dimensioni devono essere multipli di tale valore (i 12 bit inferiori di CX devono essere uguali a zero).

```
mov    bx, 2                    ; BX:CX = dimensioni del blocco
mov    cx, 0
call large_allocate
jc      alloc:err
```

Se durante l'allocazione della memoria si verifica un errore, il flag carry verrà impostato a 1 e in AX si troverà il codice d'errore (per un elenco completo dei codici d'errore, consultare le specifiche DPMI). Nell'allocazione della memoria possono verificarsi i seguenti errori:

8012h linear memory unavailable
8013h physical memory unavailable
8014h backing store unavailable
8016h no more handles
8021h invalid size (BX:CX=0)

La deallocazione di un blocco di memoria richiede che venga deallocato prima il selettore e quindi la memoria. Tale operazione può essere eseguita dalla procedura large_deallocate contenuta nel Listato 18.2.

18.9 Prestazioni del codice a 32 bit

La misurazione delle prestazioni di codice a 32 bit non è differente rispetto al codice a 16 bit, tranne per il fatto che si deve eseguire il link della libreria TIMER32.LIB (fornita su disco). Si tratta di una libreria a 32 bit per il modello small che può funzionare

con l'esempio presentato nel Listato 18.1. La procedura do_test del listato seguente mostra il modo in cui è possibile misurare la velocità di varie versioni di una procedura.

(Listato 18.3)

```
do_test proc

; alloca la memoria per il test (due stringhe da 68 KB)

        mov     bx, 2               ; BX:CX dimensioni del blocco
        mov     cx, 2000h
        call    large_allocate
        jc      dt_exit

        mov     data_selector1, ax
        mov     data_handle1, di
        mov     data_handle1[2], si

        call    test_init
        call    timing_test

        mov     bx, data_selector1
        mov     di, data_handle1
        mov     si, data_handle1[2]
        call    large_deallocate


dt_exit:

        ret

do_test endp


public test_init
test_init proc

        push    ds
        push    es

        mov     ds, ax
        mov     es, ax

; riempie di dati la prima stringa

        mov     edi, 0
        mov     eax, 01020304h
        mov     ecx, (68*1024/4)-2
        rep     stosd
        xor     eax, eax
        stosd
        stosd
```

```
        pop     es
        pop     ds
        ret

test_init endp


public timing_test
timing_test proc

        mov     esi, 0
        mov     edi, 68*1024
        push    ds
        push    es
        call    timer_on
        mov     es, data_selector1
        mov     ds, data_selector1
        call    str_copy1
        pop     es
        pop     ds
        call    timer_off
        call    timer_show

        mov     esi, 0
        mov     edi, 68*1024
        push    ds
        push    es
        call    timer_on
        mov     es, data_selector1
        mov     ds, data_selector1
        call    str_copy2
        pop     es
        pop     ds
        call    timer_off
        call    timer_show

        mov     esi, 0
        mov     edi, 68*1024
        push    ds
        push    es
        call    timer_on
        mov     es, data_selector1
        mov     ds, data_selector1
        call    str_copy2w
        pop     es
        pop     ds
        call    timer_off
        call    timer_show

        mov     esi, 0
        mov     edi, 68*1024
```

```
        push    ds
        push    es
        call    timer_on
        mov     es, data_selector1
        mov     ds, data_selector1
        call    str_copy4
        pop     es
        pop     ds
        call    timer_off
        call    timer_show

        ret

timing_test endp

align 16
public str_copy1
str_copy1 proc                  ; copia la stringa 1 byte per volta

        lbl1:
        mov     al, [esi]       ; 1     1
        inc     esi             ; 0     1
        mov     [edi], al       ; 1     1
        inc     edi             ; 0     1
        cmp     al, 0           ; 1     1
        jne     lbl1            ; 0     3       8 su un 486
        exit1:                  ; —
                                ; 3 cicli — 3 cicli per byte
        ret

str_copy1 endp

align 16
public str_copy2
str_copy2 proc                  ; copia di stringhe 2 byte per volta

        lbl2:
        mov     ax, [esi]       ; 2     2
        add     esi, 2          ; 0     1
        mov     [edi], ax       ; 2     2
        add     edi, 2          ; 0     1
        cmp     al, 0           ; 1     1
        jz      exit2           ; 0     1
        cmp     ah, 0           ; 1     1
        jne     lbl2            ; 0     3       12... 6 su un 486
        exit2:                  ; —
                                ; 6 cicli — 3 cicli per byte
        ret

str_copy2 endp
```

```
align 16
public str_copy2w
str_copy2w proc; copia di stringhe 2 byte per volta

        lbl2w:
        mov     eax, [esi]      ; 1   1
        add     esi, 2          ; 0   1
        mov     [edi], ax       ; 2   2
        add     edi, 2          ; 0   1
        cmp     al, 0           ; 1   1
        jz      exit2w          ; 0   1
        cmp     ah, 0           ; 1   1
        jne     lbl2w           ; 0   3       11... 5.5  su un 486
        exit2w:                 ; —
                                ; 5 cicli — 2.5 cicli per byte
        ret

str_copy2w endp

align 16
public str_copy4
str_copy4 proc                  ; copia di stringhe 4 byte per volta

        lbl4:
        mov     eax, [esi]      ; 1   1
        add     esi, 4          ; 0   1
        mov     [edi], eax      ; 1   1
        add     edi, 4          ; 0   1
        cmp     al, 0           ; 1   1
        jz      exit4           ; 0   1
        cmp     ah, 0           ; 1   1
        jz      exit4           ; 0   1
        bswap   eax             ; 1   1       (per il 386 sostituire con: ROR EAX, 16)
        cmp     al, 0           ; 1   1
        jz      exit4           ; 0   1
        cmp     ah, 0           ; 1   1
        jne     lbl4            ; 0   3       15... 3.75 per un 486
        exit4:                  ; —
                                ; 7 cicli — 1.75 cicli per byte

        ret

str_copy4 endp
```

La Tabella 18.2 mostra i risultati derivati dal Listato 18.3. Questi risultati sono influenzati da due importanti fattori. Innanzitutto, poiché non è possibile disabilitare con facilità gli interrupt in modalità protetta, i risultati non sono precisi come per la modalità reale. In secondo luogo, questo test copia stringhe di dimensioni maggiori rispetto alla memoria cache. Questo significa che, quando si dovranno leggere i dati dalla memoria, vi sarà una penalizzazione di due cicli sul 486 e di tre cicli sul Pentium poiché i dati da leggere non sono contenuti nella memoria cache. Su un 386 la

penalizzazione varia poiché la cache è esterna e può essere realizzata in vari modi. Inoltre vi possono essere dei ritardi anche quando si riempiono i buffer di scrittura. La prima parte della Tabella 18.2 mostra i risultati (in microsecondi) e i cicli per byte copiato. La seconda porzione della Tabella 18.2 presenta i cicli minimi teorici per byte copiato. Nel caso in cui l'operazione di copia operi su un numero di byte maggiore rispetto alle dimensioni della memoria cache, i cicli per byte non variano che di qualche punto percentuale. In questo esempio viene considerata una stringa lunga 68 KB.

18.10 Cloaking Developers Toolkit

Un altro modo interessante per scrivere programmi che operino in modalità protetta consiste nell'impiego del Cloaking Developers Toolkit prodotto da Helix Software, i produttori del gestore di memoria NetRoom. Grazie a questo toolkit è possibile sviluppare un programma "nascosto" (cloaking). Questo toolkit, dedicato principalmente alla realizzazione di driver e programmi residenti (TSR), funziona verificando ed eseguendo il codice in modalità protetta, anello 0. Le applicazioni DPMI funzionano all'anello 3, il più basso livello di privilegio. L'anello 0 è il livello di privilegio più elevato e fornisce prestazioni non disponibili in altri ambienti come, ad esempio, il DPMI. Ma si tratta anche di un metodo pericoloso poiché in caso di gravi bug si corre il rischio di bloccare il sistema. Ma ciò non è molto diverso rispetto alla programmazione DOS in modalità reale dove accade esattamente la stessa cosa.

I programmi "nascosti" che gestiscono gli interrupt, come molti driver e programmi residenti, occupano una piccola area di memoria convenzionale. Quando si verifica un interrupt, quest'area di 11 byte attiva il programma host che richiama il codice in modalità protetta. Il programma host può essere il gestore di memoria NetRoom o un driver fornito da Helix che offre driver distinti per DOS e Windows.

Nel 1986, quando vennero prodotti i primi sistemi 386, sembrò che fosse giunta la fine delle applicazioni a 16 bit. Ma in realtà questo non avvenne così in fretta. Tuttavia la tendenza odierna consiste nella realizzazione di codice a 32 bit operante in modalità protetta.

Tabella 18.2 Prestazioni per operazioni di copia di stringhe a 32 bit (microsecondi e cicli per byte).

	Pentium-60		486-33		386-25	
str_copy1	5487	4.7	21884	10.4	67648	24.3
str_copy2	3725	3.2	14228	6.7	47516	16.6
str_copy4	2593	2.2	10947	5.2	32356	11.6

Cicli per byte minimi teorici (trovando sempre i dati nella memoria cache)

	Pentium-60	486-33	386-25
str_copy1	3	8	21
str_copy2	3	6	13.5
str_copy4	1.75	3.75	10.75

Note: cicli = (tempo * MHz) / (lunghezza-in-byte)
lunghezza-in-byte = 68 KB

Capitolo 19

Note e ottimizzazioni finali

Un libro sulla programmazione sarebbe incompleto se non presentasse qualcosa di più di qualche tecnica e una montagna di materiale di riferimento. Nei Capitoli dal 13 al 15 è stato illustrato un processo che consente di convertire algoritmi composti da un singolo flusso di istruzioni in cicli di istruzioni che sfruttano appieno l'architettura a due pipeline del Pentium. Questo capitolo è principalmente dedicato alla ricerca e all'allineamento delle istruzioni in modo da sfruttare appieno queste potenzialità.

L'Appendice D contiene un gran numero di informazioni relative alle ottimizzazioni disponibili per la famiglia di microprocessori 80x86. In questo capitolo si parlerà di alcune di queste ottimizzazioni e si cercherà di scoprire le tecniche che consentono di trovare miglioramenti analoghi. Inoltre si affronterà un altro tipo di ottimizzazione facile da ottenere: l'allineamento del codice.

19.1 Velocità o compattezza del codice?

In molti casi la scelta della migliore ottimizzazione deve bilanciare la compattezza del codice con la velocità. Molte ottimizzazioni ampiamente note presentano vantaggi in entrambi i sensi. Due delle ottimizzazioni più note e utilizzate sono l'impostazione di un registro a 0 e la moltiplicazione per 2:

```
mov   ax, 0          ; tre metodi per impostare un registro a zero
xor   ax, ax
sub   ax, ax

mov   eax, 0         ; (32 bit)
xor   eax, eax
sub   eax, eax
```

Ecco tre modi per moltiplicare per 2 un registro a 16 bit:

```
mov   bx, 2        ; moltiplica AX per 2
mul   bx

shl   ax, 1        ; anche questa istruzione moltiplica AX per 2

add   ax, ax       ; anche questa istruzione moltiplica AX per 2
```

Ecco invece quattro modi per moltiplicare per 2 un registro a 32 bit:

```
mov   ebx, 2
mul   ebx

shl   eax, 1

add   eax, eax

lea   eax, [eax+eax]
```

In modalità a 32 bit, possono essere affrontate nuove e interessanti opportunità di ottimizzazione nel senso della compattezza del codice poiché alcune istruzioni utilizzano solo dati immediati a 32 bit. Ad esempio:

```
mov   eax, 1       ; 5 byte
xr    eax, eax     ; 2 byte
inc   eax          ; 1 byte
```

Ma per il momento si tornerà ai dettagli dell'impostazione di un registro a 0. Si veda la Tabella 19.1.

Anche se vi sono più modi per eseguire l'operazione, difficilmente due istruzioni o sequenze di istruzioni producono risultati esattamente identici. Le istruzioni XOR e SUB sono un caso molto raro. Il numero di cicli di CPU e le possibilità di accoppiamento delle istruzioni sul Pentium sono identiche per tutte e tre le istruzioni. Le uniche differenze sono l'effetto sui flag e le dimensioni del codice.

La Tabella 19.2 presenta i dettagli dei diversi modi per moltiplicare per 2 un registro.

Come si può vedere, nessuno dei metodi è uguale a un altro. Dunque ogni sequenza di istruzioni potrà essere sfruttata in base alle proprie esigenze.

In genere, per determinare le sequenze di istruzioni ottimali si possono considerare i seguenti fattori:

- confrontare gli effetti di due o più alternative in una determinata situazione;

Tabella 19.1 I dettagli dei diversi modi per azzerare un registro.

	Dim. codice	Modifica i flag	Accoppiabilità	Cicli		
				Pentium	**486**	**386**
mov	5	nessuno	UV	1	1	2
xor	2	tutti gli aritmetici	UV	1	1	2
sub	2	tutti gli aritmetici	UV	1	1	2

- determinare le dimensioni del codice prodotto;
- contare i cicli di CPU per ogni modello di microprocessori;
- controllare le prestazioni delle sequenze di istruzioni su ogni CPU utilizzata;
- verificare le prestazioni delle sequenze di istruzioni all'interno del codice per ogni CPU utilizzata.

Questi passi possono dire molte cose. Innanzitutto si può identificare se due sequenze di codice producono gli stessi risultati o almeno gli stessi risultati nella propria situazione. In secondo luogo, si possono determinare le dimensioni del codice a seconda dell'istruzione scelta. Si tratta di fattori importanti, come si vedrà in seguito.

Un altro e più complicato esempio consiste nel moltiplicare un registro per 2 ottenendo risultati identici. Le istruzioni MUL e SHL modificano i flag. Ma soprattutto non producono lo stesso risultato. MUL accetta operandi a 8, 16 o 32 bit e restituisce un risultato a 16, 32 o 64 bit. SHL accetta operandi delle stesse dimensioni ma restituisce anche risultati delle stesse dimensioni. Per ottenere lo stesso risultato per valori iniziali piuttosto grandi il flag carry deve essere fatto scorrere in un registro azzerato. Dovendo impiegare SHL per valori molto estesi, si dovrebbe utilizzare uno di questi esempi:

```
mov   ah, 0   ; 8->9 bit (codice a 16 o 32 bit)
shl   ax, 1

xor   dx, dx  ; 16->17 bit (codice a 16 bit)
shl   ax, 1
rcl   dx, 1

and   eax, 0FFFFh     ; 16->17 bit (codice a 32 bit)
shl   eax, 1

xor   edx, edx; 32->33 bit (codice a 32 bit)
shl   eax, 1
rcl   edx, 1
```

Naturalmente, questo significa che in realtà occorre conoscere esattamente i dati su cui si opera (e non si è ancora parlato della moltiplicazione per 4 con SHL).

Ma ecco un altro esempio. In questo caso si deve far avanzare un puntatore di due unità. Vi sono tre metodi per eseguire l'operazione:

Tabella 19.2 I dettagli dei diversi modi per moltiplicare un registro per 2 (32 bit).

				Cicli		
	Dim. codice	**Modifica i flag**	**Accoppiabilità**	**Pentium**	**486**	**386**
mov/mul	5+2	CF OF	UV/NA	1/10	/131	2/9
shl	2	tutti gli aritmetici	PU	1	1	2
add	2	tutti gli aritmetici	UV	1	1	2
lea	3	nessuno	UV	1	1	2

Tabella 19.3 I dettagli dei diversi modi per far avanzare un puntatore di due unità.

	Dim. codice	Modifica i flag	Accoppiabilità	Cicli Pentium	486	386
inc/inc	2	tutti tranne CF	U	2	2	4
add	3	tutti gli aritmetici	UV	1	1	2
lea	3	nessuno	UV	1	1	2

```
add   si, 2          ; metodo add

inc   si             ; metodo inc
inc   i

lea   si, [si+2]     ; metodo lea
```

Naturalmente, il metodo INC è più compatto solo se il puntatore deve essere fatto avanzare di una o due unità. LEA ha il vantaggio di non modificare i flag. Ma LEA è molto più potente.

19.2 L'istruzione multiuso LEA

L'istruzione LEA, Load Effective Address, è una delle più importanti istruzioni di ottimizzazione. Tale istruzione può essere utilizzata per vari scopi:
somme di puntatori senza modifica dei flag;

- moltiplicazioni veloci;
- somme fra 2, 3 e 4 operandi.

L'unico svantaggio nascosto dell'istruzione LEA è il fatto che utilizza l'unità di generazione degli indirizzi (il Pentium ne ha una per ogni pipeline). Dunque è possibile che un'istruzione LEA provochi un ritardo a causa di un blocco AGI mentre l'istruzione ADD non presenta tale ritardo poiché l'esecuzione di LEA si verifica anticipatamente nella pipeline rispetto all'istruzione ADD. Tuttavia il problema si verifica solo in sequenze di codice piuttosto rare.

L'istruzione LEA è particolarmente utile nel caso di codice in modalità a 32 bit. Ecco alcuni esempi.

Moltiplicazione:

```
lea eax, [eax+eax]      ; moltiplicazione per 2
lea eax, [eax+eax*2]    ; moltiplicazione per 3
lea eax, [eax*4]        ; moltiplicazione per 4
lea eax, [eax+eax*4]    ; moltiplicazione per 5
lea eax, [eax*8]        ; moltiplicazione per 8
lea eax, [eax+eax*8]    ; moltiplicazione per 9

add eax, eax
lea eax, [eax+eax*2]    ; moltiplicazione per 6
```

```
add eax, eax
lea eax, [eax+eax*4]        ; moltiplicazione per 10

add eax, eax
lea eax, [eax+eax*8]        ; moltiplicazione per 18

mov ebx, eax
lea ebx, [ebx+ebx*9]

lea eax, [eax+ebx*2]        ; moltiplicazione per 19
mov ebx, eax
lea ebx, [ebx+ebx*8]
lea eax, [eax+ebx*4]        ; moltiplicazione per 37

mov ebx, eax
lea ebx, [ebx+ebx*8]
lea eax, [eax+ebx*8]        ; moltiplicazione per 73
```

Somma:

```
lea eax, [eax+ebx]          ; due operandi

lea eax, [ebx+ecx]          ; tre operandi
                            ; sostituisce MOV e ADD

lea eax, [ebz+ecx+4]        ; quattro operandi, 1 immediato sostituisce MOV, ADD e ADD
lea eax, [eax+1]            ; sostituisce INC
lea eax, [eax+4]            ; sostituisce la somma fra puntatori

lea eax, [eax-1]            ; sostituisce DEC
lea eax, [eax-4]            ; sostituisce la sottrazione fra puntatori
```

Allineamento di codice e dati

L'allineamento dei dati è più difficile di quello che possa sembrare. Può essere semplice allineare i dati alle word poiché i segmenti vengono automaticamente allineati a word. Per assicurare l'allineamento a word anche dei dati, si deve inserire la direttiva EVEN prima della dichiarazione di un oggetto:

```
even                  ; allineamento a una word pari
dw 100 dup(0)         ; 100 word posizionate su word pari
```

L'allineamento a dword non è altrettanto semplice. Per eseguire un allineamento a qualsiasi intervallo che sia una potenza di 2 si utilizza la direttiva ALIGN. Il codice seguente dovrebbe allineare a dword un array di dword:

```
align 4               ; allineamento a dword
dd 100 dup(0)         ; 100 dword posizionate su dword
```

Tuttavia vi sono anche altri fattori che influenzano questo tipo di allineamento. Perché funzioni, il segmento deve essere dichiarato con allineamento a dword o supe-

riore (vedere la Tabella 18.1). L'allineamento standard è PARA (paragrafo) ovvero a 16 byte. Utilizzando le direttive di segmentazione semplificata, il tipo di allineamento è a word sia per il segmento di codice che per il segmento dati. Utilizzando le direttive di segmentazione semplificata, l'assembler MASM 5.1 consente di specificare un allineamento con la direttiva ALIGN (un allineamento maggiore dell'allineamento standard del segmento ovvero word). Ma questo funziona solo per programmi costituiti da un unico modulo oppure quando la direttiva ALIGN si trova nel primo modulo di cui viene eseguito il link. In altri casi il funzionamento di tale direttiva è incerto: nel caso in cui non possa essere garantito l'impiego della direttiva ALIGN richiesta, gli assembler TASM e MASM 6.0 e successivi generano un errore.

È anche possibile utilizzare insieme direttive di segmentazione semplificata e direttive di dichiarazione di tipo SEGMENT in modo da assicurare il corretto allineamento. Ma è meglio non fare affidamento su questo metodo se si intende aggiornare il proprio assembler. Per dichiarare un segmento dati allineato a paragrafi si può procedere nel seguente modo:

```
_data segment public para 'data'

data1 dw 0              ; dati

align 16
data2 dd 100 dup(0)     ; dati allineati a paragrafo

data ends
```

Dopo aver dichiarato correttamente i segmenti, l'allineamento dei dati può essere ottenuto in vari modi, ovvero tramite:

- dichiarazione di dati globali con allineamento opportuno;
- utilizzo di strutture con allineamento opportuno;
- allineamento di grandi operazioni su stringhe utilizzando l'allineamento tramite indirizzi all'interno del codice (discusso fra breve).

19.3 Variabili locali sullo stack

L'allocazione di spazio dello stack per le variabili locali che devono essere allineate può essere molto difficile. In modalità reale e in modalità protetta a 16 bit, si può assumere che il puntatore allo stack (SP) sia allineato alle word anche se questo in realtà non è sicuro. Se necessario è però possibile manipolare il registro BP in modo da ottenere l'allineamento:

```
push   bp

mov    bp, sp
sub    sp, local_space + 4

; (push dei registri richiesti)
```

```
; (accesso ai parametri passati)

; allinea BP per le variabili locali

push    bp
mov     ax, bp   ; allinea BP a dword
and     ax, 3    ; o usare: and   bp, 0fffch
sub     bp, ax

.....

pop bp

; pop dei registri salvati

mov     sp, bp
pop     bp
ret
```

Se si programma in modalità protetta a 32 bit, ogni PUSH e POP utilizza quattro byte di spazio sullo stack. Questo è vero anche quando si devono eseguire operazioni di push e pop sui registri di segmento. Quando si deve eseguire una push di un registro di segmento, nella word alta viene inserita una serie di 0 impedendo così il disallineamento del codice a 32 bit rispetto alle dword. Tuttavia, rimane possibile eseguire operazioni di push di word utilizzando un opportuno prefisso. L'allineamento di EBP a dword in una procedura a 32 bit può essere ottenuto nel seguente modo:

```
push    ebp
mov     ebp, esp
sub     esp, local_space + 4

; (push dei registri richiesti)

; (accesso ai parametri passati)

; allinea EBP per le variabili locali

push    ebp
mov     eax, ebp; allinea EBP a dword
and     eax, 3              ; o usare: and   ebp, 0fffffffch
sub     ebp, eax

...

pop     ebp

; pop dei registri salvati

mov     esp, ebp
pop     ebp
ret
```

19.4 Misurazione e correzione del disallineamento dei dati

Ora che si è visto come allineare i dati, ci si potrebbe chiedere se ne vale veramente la pena. A tale scopo basta eseguire un semplice test. La Tabella 19.4 mostra le penalizzazioni introdotte dal disallineamento dei dati per l'istruzione REP MOVSW.

Dalla Tabella 19.4 si può dedurre che la penalizzazione per il disallineamento è piuttosto ingente (dal 50% al 100%). La massima penalizzazione si ottiene quando sia l'origine che la destinazione sono disallineate. È piuttosto semplice correggere questo caso. Sul 386 e sul 486 è sempre molto vantaggioso leggere dati allineati e scrivere dati disallineati (tranne sul 386SX in cui non vi è alcuna differenza). Questo è dovuto al fatto che la CPU deve attendere che l'operazione di lettura sia completa mentre le operazioni di scrittura su buffer possono essere completate in seguito. Sul Pentium è vero il contrario: è meglio scrivere su indirizzi allineati.

I risultati numerici esposti nellaTabella 19.4 si applicano all'istruzione REP MOVSW. Il codice seguente mostra il modo in cui è possibile allineare SI per l'istruzione REP MOVSW:

(Listato 19.1)

```
        test    si, 1       ; verifica di indirizzo dispari
        jz      ok
        movsb               ; sposta un byte in SI dispari
        dec     cx          ; riduce il conteggio delle word
        jz      mv          ; salta solo se vi è solo 1 word da spostare
        stc                 ; CF =1 per inizio sul dispari
ok:
        rep     movsw       ; sposta le word
        jnc     done
        mv:
        movsb               ; sposta il byte dispari finale
done:
```

Gli stessi test possono essere eseguiti sull'istruzione REP MOVSD, come illustrato nella Tabella 19.5.

Anche in questo caso, i risultati esposti nella Tabella 19.5 mostrano che allineando il registro di origine (SI o ESI) è possibile ridurre la penalizzazione di disallineamento. La massima penalizzazione si ottiene nelle operazioni sulle stringhe poiché le istruzioni sono ottimizzate in modo da utilizzare il bus della memoria alla sua capacità massima. La maggior parte delle altre operazioni non impiega così pesantemente il bus.

La Tabella 19.6 mostra la penalizzazione di disallineamento per letture e scritture indipendenti dei dati.

19.5 Allineamento del codice

Come si è visto, le prestazioni possono scendere in modo sensibile nel caso di dati disallineati ma cosa si può dire a proposito dell'allineamento del codice? In altre parole, l'allineamento di procedure e cicli a word, dword o paragrafi può migliorare le prestazioni? La risposta è affermativa. Ogni CPU 80x86 legge le istruzioni suddivi-

Tabella 19.4 Penalizzazione per il disallineamento dei dati per MOVSW.

		DI allineato	DI disallineato
Pentium: nella cache	SI allineato	0%	50%
	SI disallineato	0%	50%
Pentium: non nella cache	SI allineato	0%	45%
	SI disallineato	0%	45%
486: nella cache	SI allineato	0%	26%
	SI disallineato	36%	67%
486: non nella cache	SI allineato	0%	20%
	SI disallineato	40%	50%
386: nella cache	SI allineato	0%	8%
	SI disallineato	36%	70%
386: non nella cache	SI allineato	0%	13%
	SI disallineato	46%	57%
386SX senza cache	SI allineato	0%	73%
	SI disallineato	73%	123%

Nota: ritardi calcolati come aumento in percentuale nel tempo di esecuzione rispetto al caso in cui SI e DI siano allineati.

Tabella 19.5 Penalizzazione per disallineamento dei dati per l'istruzione REP MOVSD.

		DI allineato	DI+1	DI+2	DI+3
Pentium: nella cache	SI allineato	0%	100%	100%	100%
	SI+1	0%	100%	100%	100%
	SI+2	0%	100%	100%	100%
	SI+3	0%	100%	100%	100%
Pentium: non nella cache	SI allineato	0%	80%	80%	80%
	SI+1	0%	80%	80%	80%
	SI+2	0%	80%	80%	80%
	SI+3	0%	80%	80%	80%
486: nella cache	SI allineato	0%	36%	36%	27%
	SI+1	55%	96%	96%	96%
	SI+2	55%	96%	96%	96%
	SI+3	60%	96%	96%	96%
486: non nella cache	SI allineato	0%	18%	18%	18%
	SI+1	36%	60%	60%	60%
	SI+2	36%	60%	60%	60%
	SI+3	36%	60%	60%	60%
386: nella cache	SI allineato	0%	33%	33%	33%
	SI+1	66%	116%	116%	116%
	SI+2	66%	116%	116%	116%
	51+3	66%	116%	116%	116%

(continua)

Tabella 19.5 *(continua)*

		DI allineato	DI+1	DI+2	DI+3
386: non nella cache	SI allineato	0%	17%	17%	17%
	51+1	50%	66%	66%	66%
	SI+2	50%	66%	66%	66%
	SI+3	50%	66%	66%	66%
386SX senza cache	SI allineato	0%	25%	0%	25%
	SI+1	25%	50%	40%	66%
	SI+2	20%	40%	33%	57%
	SI+3	25%	66%	57%	66%

Nota: ritardi calcolati come aumento in percentuale nel tempo di esecuzione rispetto al caso in cui SI e DI siano allineati.

Tabella 19.6 Penalizzazione di disallineamento per letture e scritture dei dati.

	Lettura di 16 bit	Scrittura di 16 bit
Pentium: nella cache	70%	50%
Pentium: non nella cache	40%	50%
486: nella cache	20%	25%
486: non nella cache	22%	25%
386: nella cache	12%	23%
386: non nella cache	15%	23%
386SX senza cache	17%	14%

dendo la memoria in un proprio modo. L'allineamento di un'istruzione molto letta può migliorare notevolmente le prestazioni. Le unità di lettura anticipata (prefetch) delle CPU 80x86 leggono le istruzioni su limiti diversi, illustrati nella Tabella 19.7.

A causa della previsione dei salti sul Pentium, l'allineamento del codice (ad eccezione dell'allineamento a word pari) diventa meno importante. L'allineamento per il 486 è il caso peggiore da tenere in considerazione. La previsione della destinazione dei salti elimina la necessità di allineamento del codice sul Pentium poiché questo diventa un problema solo quando una sezione di codice è la destinazione ripetuta di un salto. La logica di previsione della destinazione dei salti rileverà questo fatto e inizierà la lettura anticipata nella seconda coda di lettura.

Sul 486, la penalizzazione di prestazioni può essere fino al 50%, come illustrato nel Listato 19.2.

(Listato 19.2)

```
mov     cx, 1000

align   16          ; allinea al paragrafo

rept    15          ; inserisce 15 NOP per ottenere
```

```
        nop                       ; il peggiore allineamento per il 486
        endm
loop1:
        sub     cx, 1             ; ora l'istruzione SUB è divisa
        jnz     loop1             ; fra due paragrafi
        ret

        align   16                ; buon allineamento
loop2:
        sub     cx, 1
        jnz     loop2
```

Sul 486, il primo ciclo di istruzioni del Listato 19.2 richiede il 50% di tempo in più rispetto al secondo ciclo. Il degrado di prestazioni è così evidente poiché si tratta del caso peggiore di allineamento per il 486. Il ciclo è piccolo e la prima istruzione del ciclo, SUB, è lunga 2 byte, con un byte per ogni lato dei limiti di prefetch.

Sul 386, le istruzioni vengono lette anticipatamente 4 byte per volta sui limiti di dword. Infatti, sul 386 il primo ciclo del Listato 19.2 richiede solo il 7% in più rispetto al secondo ciclo.

Il Listato 19.3 contiene un tipico ciclo con il peggiore caso di allineamento del codice.

(Listato 19.3)

```
        mov     cx, 1000
        align   16                ; allinea al paragrafo

        rept    15                ; inserisce 15 NOP per ottenere
        nop                       ; il peggiore allineamento per il 486
        endm
loop1:
        add     ax, [si]
        add     si, 2
        dec     cx
        jnz     loop1

        align   16                ; buon allineamento
loop2:
        add     ax, [si]
        add     si, 2
        dec     cx
        jnz     loop2
```

In questo esempio, il 486 richiede il 25% in più sul primo ciclo rispetto al secondo e il 386 richiede solo il 3% in più.

Dopo aver sentito tutti i fattori negativi a proposito dell'allineamento del codice sul 486, è tempo di parlare dei fattori positivi. Normalmente non è necessario allineare a paragrafo le procedure e i cicli. Le prestazioni ottimali si ottengono posizionando la prima istruzione sui limiti di un paragrafo. Inoltre, l'esecuzione del codice su un Pentium elimina completamente il problema.

Un'ultima nota sull'allineamento del codice. Talvolta si trova codice che utilizza un ciclo di ritardo come:

Tabella 19.7 Limiti di lettura anticipata.

8088/188	byte
8086/186	word
80286	word
80386SX	word
80386	dword
80486	paragrafo (dimensioni di un blocco di cache)
Pentium	32 byte (dimensioni di un blocco di cache)

```
        mov   cx, 100
delay:
        loop delay
```

Il fatto che l'istruzione LOOP si trovi o meno sui limiti di prefetch fa in modo che il ritardo possa variare in modo consistente (dell'8% sul 386 e del 30% sul 486). Utilizzando la direttiva EVEN si elimina questa variabilità.

```
        mov   cx, 100
        even
delay:
        loop delay
```

19.6 Conclusione

In questo libro si è parlato di molte cose: dalla storia dei microprocessori 80x86 partendo da quelle strane istruzioni che avevano lo scopo di garantire la compatibilità con l'8080 fino ai test di prestazioni e alla programmazione numerica per il Pentium. Probabilmente il lettore ha appreso alcune cose che non conosceva. In particolare si è visto che anche dopo aver ottimizzato il codice vi è spazio per ulteriori miglioramenti legati all'uso di una o più delle seguenti tecniche:

- posizionamento corretto delle istruzioni per la programmazione superscalare;
- scelta accorta delle istruzioni;
- allineamento dei dati;
- allineamento del codice di cicli e procedure;
- test e misurazione del codice.

Dopo essere diventati esperti dell'architettura superscalare del Pentium, si sarà interessati a conoscere altre architetture superscalari per vedere ciò che riserva il futuro. Nell'ultimo capitolo ci si occuperà del PowerPC in confronto al Pentium.

Parte sesta

ESEMPI DI ARCHITETTURE SUPERSCALARI

Capitolo 20

Il PowerPC e il Pentium

IBM, Apple e Motorola si sono riunite all'inizio degli anni novanta per formare una partnership che ha dato origine a una nuova famiglia di microprocessori RISC: il PowerPC. IBM e Motorola hanno collaborato nella progettazione del chip mentre IBM ed Apple vendono sistemi completi che si basano su questa nuova architettura.

Il progetto del PowerPC si basa su un progetto IBM per la workstation RS-6000. Sono in molti a scommettere sul futuro successo del PowerPC. In questo capitolo verranno discusse le funzionalità tecniche del PowerPC nei confronti del Pentium e le future direzioni di sviluppo.

20.1 Che cosa significa RISC?

La sigla RISC sta per Reduced Instruction Set Computing. Si definisce in questo modo una CPU che è dotata di un numero di istruzioni piuttosto ridotto. In realtà i computer RISC sono abbastanza diversi.

Alla fine degli anni 70 i progettisti di computer compresero che anche se le CPU stavano diventando sempre più complesse, la maggior parte del software spendeva

l'80% del tempo di esecuzione per richiamare il 20% delle istruzioni disponibili. Questo portò allo sviluppo di una teoria in base alla quale era possibile realizzare una CPU molto veloce e potente con meno istruzioni altamente ottimizzate.

Le CPU RISC sarebbero state anche più piccole (e pertanto meno costose da produrre) e avrebbero portato alla realizzazione di sistemi di costo inferiore. Queste idee passarono al decennio successivo ma non vennero mai attuate. Sul mercato in realtà apparvero macchine che erano più economiche rispetto ai mini-computer ma erano comunque molto più costose di un PC. I sistemi RISC coprirono dunque il mercato delle workstation di fascia alta (rispetto ai PC) in una gamma di prezzi dai diecimila ai centomila dollari.

20.2 Che cosa significa CISC?

Il termine CISC o Complex Instruction Set Computing (ovvero computer ad istruzioni complesse), è stato coniato dai sostenitori dell'architettura RISC per descrivere quelle CPU per mainframe (ad esempio l'IBM 370), per mini-computer (ad esempio il DEC VAX) e microprocessori come l'80x86 e il 680x0.

20.3 Ma che cosa significa veramente RISC?

In realtà non viene ridotto nulla. Se si osserva il set di istruzioni di un microprocessore RISC, si trovano più di 100 istruzioni. Le vere caratteristiche di ciò che i progettisti chiamano processore RISC sono:

- un'architettura di tipo "carica/salva";
- un set di istruzioni altamente regolare con gestione tramite pipeline;
- un elevato numero di registri;
- registri, bus dati e bus indirizzi ad almeno 32 bit.

Un'architettura "carica/salva" consiste nel fatto che le operazioni sui dati (somme, sottrazioni, confronti e così via) prevedono istruzioni distinte per l'inserimento dei dati nei registri e per il salvataggio dei dati in memoria. Per manipolare dati in memoria sono dunque richieste due o tre istruzioni mentre, ad esempio, i processori 80x86 possono eseguire l'operazione con una singola istruzione.

Le istruzioni RISC sono molto regolari poiché quasi tutte hanno lo stesso formato in cui determinati bit specificano determinate operazioni. Grazie alla regolarità del set di istruzioni, è sufficiente una logica di decodifica relativamente semplice per gestire tramite pipeline il flusso delle informazioni. Analogamente è anche più facile realizzare processori superscalari.

In genere le CPU RISC hanno molti registri. Il PowerPC ha 32 registri generali a 32 bit e 32 registri in virgola mobile a 64 bit (al contrario i processori 80x86 ne hanno 8 per tipo e uno è il puntatore allo stack).

20.4 Dunque che cosa è meglio, un RISC o un CISC?

Dal punto di vista della scienza dei computer, la risposta è chiara. Ogni nuovo progetto sviluppato in quest'ultimo decennio è di tipo RISC e i più famosi sono Sun SPARC, MIPS, Hewlett Packard PA, DEC Alpha e PowerPC. Al contrario, dal punto di vista delle fette di mercato, le famiglie 80x86 e 680x0 hanno notevolmente sorpassato i sistemi RISC grazie alla base hardware installata, all'ampia varietà di software, al basso costo dei sistemi e alla competitività del mercato consumer. Se alla fine degli anni settanta i componenti per computer (memoria, dischi, monitor e così via) fossero stati più costosi, l'esplosione di vendite di PC degli anni 80 sarebbe stata ritardata di cinque anni e si sarebbero probabilmente venduti 50 milioni di chip RISC all'anno.

Ma i sistemi CISC presentano alcuni punti tecnici di vantaggio. Poiché i sistemi RISC hanno bisogno di eseguire più istruzioni semplici per ottenere il risultato di un'unica istruzione CISC, i programmi RISC tendono ad essere più estesi rispetto ai corrispondenti programmi CISC. In genere questo aumento dimensionale può raggiungere il 50%. Se i programmi sono più estesi, ci sarà bisogno di più memoria e di più cache. Ciò che ha spinto a realizzare istruzioni complesse negli anni 60 e 70 era la preoccupazione di ridurre il numero di istruzioni da leggere nella CPU poiché il tempo di lettura dell'istruzione era molto lungo in confronto al tempo di esecuzione.

Nei sistemi multitasking è fondamentale il tempo necessario per eseguire una commutazione di contesto. Uno dei fattori più importanti in questo senso è il numero di registri che devono essere salvati. Il PowerPC ha circa tre volte il numero di registri rispetto al Pentium. Dunque è strano che i sistemi RISC siano stati ampiamente utilizzati nel mercato delle workstation dove utilizzano UNIX, un sistema operativo multitasking e che i sistemi 80x86, storicamente, siano stati perlopiù impiegati per applicazioni DOS mono-utente e per il multitasking non-preemptive di Windows.

20.5 Il Pentium è un RISC o un CISC?

Quando Intel annunciò il 486 affermò che nello sviluppo del nuovo chip le componenti principali del 386 erano state completamente ridisegnate. Le operazioni di base venivano eseguite da un nucleo RISC e le istruzioni complesse venivano gestite da una logica distinta. Il 486 era gestito a pipeline e tutte le istruzioni semplici richiedevano un solo ciclo. Con il Pentium si è dimostrato che questo sottoinsieme di istruzioni può essere eseguito in parallelo in due pipeline. Dunque probabilmente il Pentium è meglio sia di un CISC che di un RISC, tranne per il fatto che, come si vedrà più avanti, il numero dei registri del Pentium può essere un fattore limitante.

20.6 Microprocessori superscalari

La gestione tramite pipeline consente a una CPU di eseguire le più semplici operazioni in un unico ciclo. Il modo più efficace per migliorare le prestazioni in un micropro-

cessore pipeline consiste nel moltiplicare le pipeline. Questa tecnica è nota con il nome di architettura superscalare. Quando il microprocessore ha un'unica pipeline, si parla di architettura scalare. Le architetture scalari sono il contrario delle architetture parallele. In generale, le macchine ad architettura parallela sono adatte a problemi che utilizzano più processori (da decine a migliaia) i quali lavorano sullo stesso problema con porzioni di dati indipendenti. Le architetture superscalari hanno lo stesso ambito di lavoro delle architetture scalari: un singolo flusso di istruzioni operanti su un unico flusso di dati.

Il concetto delle pipeline multiple è semplice. Raddoppiando il numero delle pipeline, si possono raddoppiare le prestazioni globali del sistema. Avendo a disposizione 4 pipeline si otterrà più o meno il quadruplo delle prestazioni. Come si è visto dagli esempi dei capitoli precedenti, per ottenere un raddoppio delle prestazioni è necessario rielaborare completamente anche i cicli di istruzioni più semplici. Ma spesso i risultati di alcune istruzioni devono essere utilizzati da un'istruzione successiva. Quando la distanza media fra queste dipendenze è piuttosto ampia, possono funzionare contemporaneamente più pipeline.

Ma dunque vi è un limite implicito al numero di pipeline che possono essere utilizzate con efficacia in una macchina superscalare. Normalmente i ricercatori sono concordi nel pensare che il limite pratico sia costituito da sei pipeline. Interi libri sono dedicati alle discussioni riguardanti questo limite. In parole semplici, il limite di sei pipeline si basa sul fatto che le operazioni della CPU su applicazioni generali devono attendere i risultati di operazioni precedenti. Il numero di operazioni indipendenti dai dati che possono essere eseguite simultaneamente, almeno per gli algoritmi più utilizzati è proprio di 6. Alcuni algoritmi possono utilizzare più pipeline ma altri meno (curiosamente Intel ha chiamato le pipeline del Pentium U e V lasciando disponibili le lettere W, X, Y e Z).

Nella progettazione di un microprocessore occorre tenere in considerazione vari fattori. Ad esempio è meglio utilizzare 4 pipeline e 256 KB di cache o 6 pipeline e 32 KB di cache? Può sembrare un esempio non significativo ma il numero di algoritmi che utilizzano cinque o sei pipeline è piuttosto ridotto e l'aumento di prestazioni insignificante rispetto a 4 pipeline con una cache più estesa.

Questo problema è simile a quello che si presenta quando si devono ottimizzare programmi per macchine DOS e Windows. Conviene avere 2 MB di cache per il disco e 14 MB per l'esecuzione dei programmi o 6 MB di cache per il disco e 10 MB per il software? Molto dipende dal tipo di software utilizzato. Si può scoprire la differenza riavviando il sistema e provando con un'altra configurazione fino a trovare quella ottimale. Ma una volta progettato, un chip non può essere modificato: bisogna progettare un nuovo chip.

A mano a mano che avanza la tecnologia, diventa possibile inserire sempre più transistor in un unico chip. La legge di Moore stabilisce che il numero dei transistor su un chip raddoppia ogni 18 mesi. E anche se non si tratta di una legge fisica, è un fenomeno osservabile nella storia tecnologica degli ultimi 25 anni. Sembrerebbe dunque che la prossima generazione di CPU avrà 6 pipeline e 512 KB di cache. Ma probabilmente vi è un punto di incontro ottimale fra flusso dei dati tra CPU e memoria, dimensioni della cache, numero di pipeline che possono funzionare a pieno regime a una determinata velocità di clock e numero dei transistor contenuti nel chip. Grazie alla legge di Moore, il numero dei transistor sembrerebbe abbastanza prevedibile. Ma

la legge di Moore non può essere applicata all'infinito. A un certo punto interverranno le leggi della fisica che impediranno la realizzazione di circuiti delle dimensioni di pochi atomi e questo probabilmente avverrà molto presto. La tecnologia corrente realizza circuiti delle dimensioni di 0.6 micron ovvero 6×10^{-7} metri. L'atomo dell'idrogeno ha un diametro approssimativo di 10^{-10} metri. Sulla base di questi numeri, basteranno 10 o 20 anni per raggiungere i limiti atomici. È piuttosto facile prevedere che i chip che appariranno sul mercato fra sei anni avranno 16 volte più transistor rispetto a quelli odierni e che tra 12 anni i chip avranno un numero di transistor pari a 256 volte quelli odierni.

20.7 Microprocessori superscalari: tecniche e terminologia

I microprocessori scalari raggiungono le loro massime prestazioni tramite l'impiego di pipeline. Le massime prestazioni di 1 ciclo per istruzione non possono essere raggiunte in altro modo (per una descrizione dettagliata si consulti il Capitolo 10). Dunque il limite è di un ciclo per istruzione. Per raggiungere prestazioni superiori occorre aggiungere una nuova pipeline. Ma il mantenimento di un livello di prestazioni di 2 o più istruzioni per ciclo richiede tecniche più complesse rispetto alla semplice aggiunta di pipeline per le seguenti limitazioni fondamentali:

- dipendenze fra i dati;
- dipendenze procedurali;
- conflitti fra le risorse.

Le dipendenze fra i dati si verificano quando i risultati di un'operazione sono necessari per l'operazione successiva. Ad esempio quando il risultato di un'operazione è necessario per far procedere l'operazione successiva si può verificare un blocco AGI (Address Generation Interlock). Vi sono dipendenze anche nel caso di letture dopo una scrittura o scritture dopo una scrittura (vedere il Capitolo 9).

Le dipendenze procedurali si verificano quando avviene un salto poiché il microprocessore deve attendere l'esecuzione del salto per conoscere la prossima istruzione che dovrà eseguire.

I conflitti di risorse possono verificarsi in varie fasi dell'esecuzione in pipeline. Questo è dovuto al fatto che un computer è composto da vari circuiti. I più importanti sono la memoria base, la memoria cache, i bus, i registri e le unità di esecuzione degli di scorrimenti e delle somme. Quando due istruzioni richiedono l'uso della stessa risorsa nello stesso ciclo si verifica un conflitto.

Non vi sono soluzioni hardware semplici per il problema della dipendenza dei dati. La soluzione principale è quella che costituisce un po' lo spirito di questo manuale ovvero una programmazione accurata. Gli altri due limiti possono essere superati grazie a particolari tecniche di progettazione hardware come ad esempio:

- il sistema di previsione della destinazione dei salti:
- la duplicazione delle risorse;

- l'esecuzione non lineare;
- la ridenominazione dei registri;
- l'esecuzione anticipata del codice che si trova alla destinazione dei salti;
- l'esecuzione speculativa;
- l'impiego di microprocessori VLIW (Very Long Instruction Word).

Per poter eseguire i salti condizionali in un unico ciclo di CPU, il Pentium utilizza la previsione della destinazione dei salti. Il Pentium infatti è in grado di superare le dipendenze implicite nei salti condizionali ignorando i dati e prevedendo la destinazione di un salto sulla base dei salti precedenti. Se la previsione si rivela errata viene introdotto un ritardo.

La duplicazione di risorse è una soluzione che consente di eliminare i "colli di bottiglia" che intervengono al momento dell'esecuzione del codice. Ad esempio, il Pentium può eseguire nello stesso ciclo due istruzioni che accedono alla memoria in quanto contiene due circuiti di accesso alla memoria.

L'esecuzione non lineare è la possibilità di un microprocessore di eseguire le istruzioni secondo un ordine diverso rispetto a quello specificato nel programma ottenendo comunque sempre lo stesso risultato. Ecco una breve descrizione di questa funzionalità avanzata. Le istruzioni vengono lette e decodificate e quindi vengono inserite in un buffer chiamato coda delle istruzioni. Il microprocessore segue costantemente le istruzioni pronte per essere eseguite e le risorse disponibili e quindi sceglie l'istruzione da eseguire. Ovviamente la coda delle istruzioni si trova nella pipeline fra la fase di decodifica e quella di esecuzione. Il microprocessore legge e decodifica le istruzioni tentando di mantenere sempre piena la coda delle istruzioni. Contemporaneamente, le istruzioni devono essere esaminate per rilevare eventuali dipendenze dai dati e, se necessario, per utilizzare un'esecuzione lineare. Una delle funzionalità principali di questo tipo di architettura è il fatto che le unità di esecuzione sono molto specializzate, ovvero sono in grado di eseguire solo istruzioni di salto o solo istruzioni di caricamento. Al contrario del Pentium il PowerPC è dotato di questa funzionalità.

La possibilità di cambiare nome ai registri è una tecnica che impedisce che i registri interni vengano resi inutilizzabili quando si impiega un'architettura a esecuzione non lineare. Ad esempio la prima istruzione potrebbe richiedere l'utilizzo di un registro come un puntatore a memoria e potrebbe rimanere in attesa nella coda delle istruzioni a causa di un conflitto nella risorsa "bus della memoria". A questo punto si potrebbe eseguire la seconda istruzione (dunque prima della prima istruzione) e tale istruzione potrà comunque modificare un registro richiesto dalla prima istruzione (una scrittura dopo una lettura) grazie alla possibilità di rinominare i registri. Questa tecnica funziona fornendo ulteriori registri che consentono di ristabilire le corrette relazioni fra registri e valori. La tecnica di ridenominazione dei registri è impiegata da alcuni processori PowerPC.

La tecnica Branch Folding consente di eliminare dalla coda delle istruzioni le istruzioni di salto incondizionato facendo continuare la lettura delle istruzioni dalla destinazione del salto. Il vantaggio è che il salto non richiede alcun ciclo di CPU. Il PowerPC, al contrario del Pentium, ha questa possibilità. I salti condizionali non possono essere gestiti in questo modo poiché l'esecuzione dipende dal risultato.

L'esecuzione speculativa consente di prevedere la destinazione di un salto e di eseguire le istruzioni che si trovano alla destinazione, presumendo che vi sarà bisogno dei risultati di tali istruzioni. Se la previsione di salto è errata, i risultati anticipati ottenuti vengono eliminati. Una tecnica più avanzata consiste nell'esecuzione di entrambe le parti di un salto condizionato. Questa funzionalità non è presente né sul Pentium né sul PowerPC.

I processori VLIW (Very Long Instruction Word) appartengono a una classe progettuale che si posiziona al di sopra della normale architettura superscalare. Le tecniche di progettazione superscalare tradizionali consentono di eseguire più istruzioni indipendenti solo quando si determina che non vi sono conflitti. Un processore VLIW esegue più operazioni concorrenti specificate in una singola istruzione. Il programmatore e/o il compilatore determina dunque le operazioni che devono essere eseguite da una singola istruzione. Tali operazioni vengono quindi codificate in un'istruzione molto lunga. La CPU può facilmente eseguire tutte le operazioni codificate poiché vengono disposte in modo da non entrare in conflitto l'una con l'altra. I processori VLIW sono in un certo senso una nuova forma di CISC tranne per il fatto che ogni istruzione è composta da molte istruzioni più piccole. Al momento attuale sia Intel che Hewlett Packard hanno annunciato l'intenzione di realizzare un processore VLIW compatibile con l'architettura RISC HP PA e con l'architettura 80x86.

20.8 Che cosa c'è in un PowerPC?

Il PowerPC è una famiglia di CPU RISC che si basa sull'architettura POWER IBM. Il PowerPC 601 è il primo chip di questa serie e viene impiegato in computer Power Macintosh. Ecco alcune delle caratteristiche del PowerPC 601:

- indirizzamento a 32 bit;
- 32 registri generali a 32 bit;
- 32 registri in virgola mobile a 64 bit;
- 3 unità di esecuzione con possibilità di esecuzione non lineare;
- 32 KB di memoria cache (comune per dati e codice).

La funzionalità più interessante è il fatto che il PowerPC 601 ha 3 unità di esecuzione che consentono di eseguire le istruzioni in modo non lineare. Ogni unità di esecuzione ha caratteristiche proprie:

- una unità per gli interi;
- una unità di elaborazione dei salti;
- una unità in virgola mobile.

Questo significa che il 601 è in grado di eseguire 3 istruzioni nello stesso ciclo sempre che una sia un salto, una sia un'istruzione in virgola mobile e la terza sia qualsiasi altra cosa (un'operazione intera). Senza eseguire alcun test, si può pensare che questo chip non sia potente come il Pentium che è in grado di eseguire nello stesso

ciclo due istruzioni intere o un'istruzione intera e un salto (con la possibilità di prevedere la destinazione dei salti). Sembrerebbe che il 601 presenti vantaggi nell'esecuzione di operazioni in virgola mobile poiché il Pentium accusa ritardi quando tenta di eseguire insieme istruzioni intere e in virgola mobile mentre il 601 può eseguire le istruzioni nello stesso ciclo.

Il vero vantaggio delle architetture non lineari è però la possibilità di aggiungere ulteriori unità di esecuzione. Il PowerPC 603 ne ha 5:

- una unità intera (IU);
- una unità di caricamento e salvataggio (LSU);
- una unità dei registri di sistema (SRU);
- una unità di elaborazione dei salti (BPU);
- una unità in virgola mobile (FPU).

L'unità LSU esegue tutte le istruzioni di caricamento e memorizzazione e i trasferimenti fra i registri generali, i registri in virgola mobile e la memoria. L'unità SRU esegue varie istruzioni di sistema come il trasferimento di dati tra i registri di sistema e i registri di condizione (analoghi ai flag dell'80x86).

Un altro vantaggio del PowerPC è legato al numero dei registri. Come si è visto nell'ottimizzazione dei programmi Pentium, normalmente si cerca di utilizzare il maggior numero possibile dei 7 registri generali disponibili. Dunque i programmi più complessi con molti cicli nidificati in genere soffrono di questa mancanza di registri sul Pentium.

20.9 Ma un PowerPC è meno costoso?

Vi sono due modi per rispondere a questa domanda. Dal punto di vista di un produttore, IBM e Motorola affermano che le dimensioni del chip sono pari a meno della metà rispetto ai primi chip Pentium (vedere la Tabella 20.1). Mantenendo invariati tutti gli altri fattori (ovvero i costi legati alla produzione) questo significa che il PowerPC potrebbe costare circa la metà di un Pentium. Dal punto di vista del ciclo di vita di un prodotto, attualmente i chip Intel hanno però alcuni vantaggi. Innanzitutto Intel vende molti più chip, circa dieci volte tanto. Questo consente di ottenere alcune economie di scala negli impianti di fabbricazione dei chip e consente di suddividere i costi di ricerca e sviluppo su più unità. Inoltre, più o meno nello stesso periodo in cui venivano forniti i primi sistemi con PowerPC 601, Intel ha iniziato la realizzazione di una seconda generazione di Pentium con circuiti delle dimensioni di 0.6 micron (rispetto agli 0.8 micron iniziali). Questo ha ridotto sia le dimensioni che i costi di produzione (vedere la Tabella 20.1).

Probabilmente la caratteristica più importante dell'architettura del PowerPC è la possibilità di utilizzare più unità di esecuzione che si occupano delle diverse classi di istruzioni. Al contrario, nell'architettura del Pentium la maggior parte delle istruzioni accoppiabili può essere eseguita in entrambe le pipeline. Questo significa che nel Pentium molte delle risorse sono duplicate prevedendone un uso simultaneo. L'architettura del PowerPC ha preso il concetto di una singola unità di esecuzione e l'ha suddiviso

Tabella 20.1 Confronto fra Pentium e PowerPC.

	Pentium	**Pentium**	**PowerPC 601**	**PowerPC 603**
Velocità (MHz)	60, 66	90, 100	66/80	60/75
Data di produzione	2° trim.. 93	1° trim. 94	3° trim 93	3° trim 94
Transistor (milioni)	3.1	3.3	2.8	1.6
Dimensioni elementi (micron)	0.8	0.6	0.6	0.5
Tecnologia	BICMOS	BICMOS	CMOS	CMOS
Dimensioni (mm2)	294	163	121	85

in più unità più piccole. Questa tecnica riduce le dimensioni globali del chip poiché vi è una minore duplicazione di risorse. Inoltre questo tipo di progetto consente di espandere agevolmente le prestazioni e/o di duplicare alcune unità di esecuzione. Probabilmente questo concetto consente di produrre chip meno costosi. Non è chiaro, invece, se questo solo concetto consenta di realizzare chip più potenti allo stesso prezzo. Infatti occorre sempre tenere in considerazione la legge di Moore: in 6 anni il numero di transistor nei chip viene moltiplicato per 16.

20.10 Caratteristiche dei futuri microprocessori

È importante notare che tutte le tecniche superscalari di cui si è parlato non sono specifiche di una determinata architettura. Ad esempio, è possibile che i prossimi processori Intel prevedano l'esecuzione non lineare con 3 unità di esecuzione intere, un'unità di elaborazione dei salti e un'unità per operazioni in virgola mobile. Analogamente, un PowerPC potrà essere dotato di due unità di caricamento/memorizzazione e due unità intere. Inoltre al Pentium potrebbe essere aggiunta la funzione di esecuzione anticipata del codice di destinazione dei salti (Branch Folding) mentre al PowerPC potrebbe essere aggiunta la funzione di previsione della destinazione dei salti. L'unico limite principale è il numero dei registri contenuti nei microprocessori della famiglia 80x86. Non vi è alcun modo semplice per superare questo limite. Si possono aggiungere registri di utilizzo generale, ad esempio introducendo nuove istruzioni e/o estensioni alle modalità di indirizzamento. Un altro metodo consiste nell'impiegare un bit nel segmento di codice per attivare l'uso di un set di istruzioni modificato. Questi metodi hanno però lo svantaggio di non garantire la piena compatibilità con il software meno recente. Ma raddoppiare la densità di transistor in un chip richiede solo 18 mesi. La riscrittura del software richiede molto più tempo. D'altra parte, IBM sta elaborando una versione del PowerPC (615) contenente hardware per l'esecuzione del set di istruzioni 80x86.

I benchmark per il PowerPC in genere confrontano codice ottimizzato per PowerPC e vecchio codice 80x86 eseguito sul Pentium. Come si è visto in questo manuale è però possibile migliorare le prestazioni del codice dal 100% al 500%. E molte delle routine

come quelle presentate in questo manuale sono ampiamente utilizzate nei programmi commerciali. Infine non è vero che la riscrittura del vecchio codice 80x86 rappresenti uno svantaggio poiché il PowerPC parte con una base di software installato molto ridotta.

L'industria dei computer deve però sempre tenere in considerazione le esigenze dei clienti che chiedono sempre più velocità ma anche compatibilità.

Vi sono alcune funzionalità che Intel potrebbe aggiungere alle prossime generazioni di chip in modo da ottenere prestazioni analoghe al PowerPC. Ma, a parte questo, i progettisti Intel dovranno trovare nuove strade per mantenere sempre competitiva l'architettura 80x86.

Appendice A

Il set di istruzioni

Questa appendice è suddivisa in due sezioni. La prima contiene le istruzioni intere mentre la seconda contiene le istruzioni in virgola mobile.

A.1 Set di istruzioni 80x86 (8088 - Pentium)

Questa sezione include la lunghezza e i tempi di esecuzione di tutte le istruzioni (escluse quelle in virgola mobile) e informazioni specifiche di accoppiamento di istruzioni per il Pentium. Per informazioni sulle abbreviazioni utilizzate, si consulti la legenda posta al termine di questa sezione.

AAA	**Ascii Adjust after Addition**						
	byte	**8088**	**186**	**286**	**386**	**486**	**Pentium**
aaa	1	8	8	3	4	3	3 NP
AAD	**Ascii Adjust ax before Division (il divisore è il secondo byte)**						
	byte	**8088**	**186**	**286**	**386**	**486**	**Pentium**
aad	2	60	15	14	19	14	10 NP
AAM	**Ascii Adjust ax after Multiply (il divisore è il secondo byte)**						
	byte	**8088**	**186**	**286**	**386**	**486**	**Pentium**
aam	2	83	19	16	17	15	18 NP
AAS	**Ascii Adjust al after Subtraction**						
	byte	**8088**	**186**	**286**	**386**	**486**	**Pentium**
aas	1	8	7	3	4	3	3 NP

ADC	integer ADd with Carry						
	byte	**8088**	**186**	**286**	**386**	**486**	**Pentium**
adc reg, reg	2	3	3	2	2	1	1 PU
adc mem, reg	2+d(0,2)	24+EA	10	7	7	3	3 PU
adc reg, mem	2+d(0,2)	13+EA	10	7	6	2	2 PU
adc reg, imm	2+i(1,2)	4	4	3	2	1	1 PU
adc mem, imm	2+d(0,2) +i(1,2)	23+EA	16	7	7	3	3 PU*
adc acc, imm	1+i(1,2)	4	4	3	2	1	1 PU
	* = non accoppiabili se vi è uno scostamento e una costante						

ADD	integer ADDition						
	byte	**8088**	**186**	**286**	**386**	**486**	**Pentium**
add reg, reg	2	3	3	2	2	1	1 UV
add mem, reg	2+d(0,2)	24+EA	10	7	7	3	3 UV
add reg, mem	2+d(0,2)	13+EA	10	7	6	2	2 UV
add reg, imm	2+i(1,2)	4	4	3	2	1	1 UV
add mem, imm	2+d(0,2)+i(1,2)	23+EA	16	7	7	3	3 UV*
add acc, imm	1+i(1,2)	4	4	3	2	1	1 UV
	* = non accoppiabili se vi è uno scostamento e una costante						

AND	logical AND						
	byte	**8088**	**186**	**286**	**386**	**486**	**Pentium**
and reg, reg	2	3	3	2	2	1	1 UV
and mem, reg	2+d(0,2)	24+EA	10	7	7	3	3 UV
and reg, mem	2+d(0,2)	13+EA	10	7	6	2	2 UV
and reg, imm	2+i(1,2)	4	4	3	2	1	1 UV
and mem, imm	2+d(0,2) +i(1,2)	23+EA	16	7	7	3	3 UV *
and acc, imm	1+i(1,2)	4	4	3	2	1	1 UV
	* = non accoppiabili se vi è uno scostamento e una costante						

ARPL	Adjust RPL field of selector (286+)				
	byte	**286**	**386**	**486**	**Pentium**
arpl reg, reg	2	10	20	9	7 NP
arpl mem, reg	2+d(0-2)	11	21	9	7 NP

BOUND	check array index against BOUNDs (186+)					
	byte	**186**	**286**	**386**	**486**	**Pentium**
bound reg, mem	4	35	13	10	7	8 NP

BSF	Bit Scan Forward (386+)			
	byte	**386**	**486**	**Pentium**
bsf r16, r16	3	10+3n	6-42	6-34 NP
bsf r32, r32	3	10+3n	6-42	6-42 NP
bsf r16, m16	3+d(0,1,2)	10+3n	7-43	6-13 NP
bsf r32, m32	3+d(0,1,2,4)	10+3n	7-43	6-43 NP

BSR	**Bit Scan Reverse (386+)** byte	**386**	**486**	**Pentium**
bsr r16, r16	3	10+3n	6-103	7-39 NP
bsr r32, r32	3	10+3n	7-104	7-71 NP
bsr r16, m16	3+d(0,1,2)	10+3n	6-103	7-40 NP
bsr r32, m32	3+d(0,1,2,4)	10+3n	7-104	7-72 NP

BSWAP	**Byte SWAP (486+)** byte	**486**	**Pentium**
bswap r32	2	1	1 NP

BT	**Bit Test (386+)** byte		**386**	**486**	**Pentium**
bt reg, reg	3		3	3	4 NP
bt mem, reg	3+d(0,1,2,4)		12	8	9 NP
bt reg, imm8	3+i(1)		3	3	4 NP
bt mem, imm8	3+d(0,1,2,4)+i(1)	6	3	4 NP	

BTC	**Bit Test and Complement (386+)** byte	**386**	**486**	**Pentium**
btc reg, reg	3	6	6	7 NP
btc mem, reg	3+d(0,1,2,4)	13	13	13 NP
btc reg, imm8	3+i(1)	6	6	7 NP
btc mem, imm8	3+d(0,1,2,4)+i(1)	8	8	8 NP

BTR	**Bit Test and Reset (386+)** byte	**386**	**486**	**Pentium**
btr reg, reg	3	6	6	7 NP
btr mem, reg	3+d(0,1,2,4)	13	13	13 NP
btr reg, imm8	3+i(1)	6	6	7 NP
btr mem, imm8	3+d(0,1,2,4)+i(1)	8	8	8 NP

BTS	**Bit Test and Set (386+)** byte	**386**	**486**	**Pentium**
bts reg, reg	3	6	6	7 NP
bts mem, reg	3+d(0,1,2,4)	13	13	13 NP
bts reg, imm8	3+i(1)	6	6	7 NP
bts mem, imm8	3+d(0,1,2,4)+i(1)	8	8	8 NP

CALL	**CALL subroutine** byte	**8088**	**186**	**286**	**386**	**486**	**Pentium**
call near	3	23	14	7+m	7+m	3	1 PV
call reg	2	20	13	7+m	7+m	5	2 NP
call mem16	2+d(0-2)	29+EA	19	11+m	10+m	5	2 NP
call far	5	36	23	13+m	17+m	18	4 NP
call mem32	2+d(0-2)	53+EA	38	16+m	22+m	17	4 NP

CALL	CALL subroutine						
		Modalità protetta					
	byte	8088	186	286	386	486	Pentium
	byte			286	386	486	Pentium
call far	5			26+m	34+m	20	4-13 NP
call mem32	2+d(0-2)			29+m	38+m	20	5-14 NP
	Non sono mostrati i cicli per le CALL attraverso le porte call e task						

CBW	Convert Byte to Word (AL->AX)						
	byte	8088	186	286	386	486	Pentium
cbw	1	2	2	2	3	3	3 NP

CDQ	Convert Double to Quad (EAX->EDX:EAX)		
	byte	386	486	Pentium
cdq	1	2	3	2 NP

CLC	CLear the Carry flag						
	byte	8088	186	286	386	486	Pentium
clc	1	2	2	2	2	2	2 NP

CLD	CLear the Direction flag (direzione in avanti)						
	byte	8088	186	286	386	486	Pentium
cld	1	2	2	2	2	2	2 NP

CLI	CLear the Interrupt flag (disattiva gli interrupt)						
	byte	8088	186	286	386	486	Pentium
cli	1	2	2	3	3	5	7 NP

CLTS	CLear Task Switched flag in CR0 (286+)				
	byte	286	386	486	Pentium
clts	2	2	5	7	10 NP

CMC	CoMplement Carry flag						
	byte	8088	186	286	386	486	Pentium
cmc	1	2	2	2	2	2	2 NP

CMP	CoMPare two operands						
	byte	8088	186	286	386	486	Pentium
cmp reg, reg	2	3	3	2	2	1	1 UV
cmp mem, reg	2+d(0,2)	13+EA	10	7	5	2	2 UV
cmp reg, mem	2+d(0,2)	13+EA	10	6	6	2	2 UV
cmp reg, imm	2+i(1,2)	4	4	3	2	1	1 UV
cmp mem, imm	2+d(0,2)						
	+i(1,2)	14+EA	10	6	5	2	2 UV*
cmp acc, imm	1+i(1,2)	4	4	3	2	1	1 UV
	* = non accoppiabili se vi è uno scostamento e una costante						

CMPS/CMPSB/CMPSW/CMPSD	CoMPare String operands (confronta DS:[SI] con ES:[DI])						
	byte	8088	186	286	386	486	Pentium
cmpsb	1	30	22	8	10	8	5 NP
cmpsw	1				10	8	5 NP
cmpsd	1				10	8	5 NP
repX cmpsb	2	9+30n	5+22n	5+9n	5+9n	7+7n*	9+4n NP
repX cmpsw	2	9+30n	5+22n	5+9n	5+9n	7+7n*	9+4n NP
repX cmpsd	2				5+9n	7+7n*	9+4n NP

repX = repe, repz, repne o repnz
* = 5 se n=0

CMPXCHG	CoMPare and EXCHanGe (486+)		
	byte	486	Pentium
cmpxchg reg, reg	3	6	5 NP
cmpxchg mem, reg	3+d(0-2)	7-10	6 NP

CMPXCHG8B	CoMPare and EXCHanGe 8 Bytes (Pentium)	
	byte	Pentium
cmpxchg8b mem	3+d(0-2)	10 NP

CPUID	CPU IDentification (Pentium)	
	byte	Pentium
cpuid	2	14 NP

CWD	Convert Word to Double (AX->DX:AX)						
	byte	8088	186	286	386	486	Pentium
cwd	1	5	4	2	2	3	2 NP

CWDE	Convert Word to Dword (386+) (AX->EAX)			
	byte	386	486	Pentium
cwde	1	3	3	3 NP

DAA	Decimal Adjust AL after Addition						
	byte	8088	186	286	386	486	Pentium
daa	1	4	4	3	4	2	3 NP

DAS	Decimal Adjust AL after Subtraction						
	byte	8088	186	286	386	486	Pentium
das	1	4	4	3	4	2	3 NP

DEC	DECrement						
	byte	8088	186	286	386	486	Pentium
dec r8	2	3	3	2	2	1	1 UV
dec rl6	1	3	3	2	2	1	1 UV
dec r32	1	3	3	2	2	1	1 UV
dec mem	2+d(0,2)	23+EA	15		6	3	3 UV

DIV	unsigned DIVide byte	8088	186	286	386	486	Pentium
div r8	2	80-90	29	14	14	16	17 NP
div r16	2	144-162	38	22	22	24	25 NP
div r32	2				38	40	41 NP
div mem8	2+d(0-2)	86-96+EA	35	17	17	16	17 NP
div meml6	2+d(0-2)	150-168+EA	44	25	25	24	25 NP
div mem32	2+d(0-2)				41	40	41 NP

dividendo implicito	*divisore*	*quoziente*	*resto*
AX	byte	AL	AH
DX:AX	word	AX	DX
EDX:EAX	dword	LAX	EDX

ENTER	Crea spazio sullo stack per i parametri di una procedura (186+) byte	8088	186	286	386	486	Pentium
enter imml6, 0	3		15	11	10	14	11 NP
enter imml6, 1	4		25	15	12	17	15 NP
enter imml6, imm8	4		22+16n	12+4n	15+4n	17+3i	15+2i NP
	n=imm8-1; i=imm8						

ESC **ESCape**

I codici operativi di ESC da D8 a DF sono utilizzati dalle istruzioni in virgola mobile

HLT	HaLT byte	8088	186	286	386	486	Pentium
hlt	1	2	2	2	5	4	4 NP

IDIV	Integer signed DIVide byte	8088	186	286	386	486	Pentium
idiv r8	2	101-112	4452	17	19	19	22 NP
idiv rl6	2	165-184	53-61	25	27	27	30 NP
idiv r32	2				43	43	46 NP
idiv mem8	2+d(0-2)	107-118+EA		50-58	20	22	20 22 NP
idiv meml6	2+d(0-2)	171-190+EA		59-67	28	30	28 30 NP
idiv mem32	2+d(0-2)				46	44	46 NP

dividendo implicito	*divisore*	*quoziente*	*resto*
AX	byte	AL	AH
DX:AX	word	AX	DX
EDX:EAX	dword	EAX	EDX

IMUL	Integer signed MULtiply (moltiplicazione di accumulatori) byte	8088	186	286	386	486	Pentium
imul r8	2	80-98	25-28	13	9-14	13-18	11 NP
imul rl6	2	128-154	34-37	21	9-22	13-26	11 NP
imul r32	2				9-38	13-42	10 NP
imul mem8	2+d(0-2)	86-104+EA	32-34	16	12-17	13-18	11 NP
imul meml6	2+d(0-2)	134-160+EA		40-43	24	12-25	13-26
11 NP							
imul mem32	2+d(0-2)				12-41	13-42	10 NP

IMUL	**Integer signed MULtiply**		
	moltiplicando implicito	*moltiplicatore*	*risultato*
	AL	byte	AX
	AX	word	DX:AX
	EAX	dword	EDX:EAX

	Moltiplicazione di 2 o 3 operandi **byte**		**186**	**286**	**386**	**486**	**Pentium**
imul rl6, imm	2+i(1,2)			21	9-14/ 9-22	13-18/ 13-26	10 NP
imul r32, imm	2+i(1,2)				9-38	13-42	10 NP
imul rl6,rl6,imm	2+i(1,2)		22/29	2	9-14/ 9-2	13-18/ 13-26	10 NP
imul r32,r32,imm	2+i(1,2)				9-38	13-42	10 NP
imul rl6,ml6,imm	2+d(0-2) + i(1,2)		25/32	2	12-17/ 12-25	13-18/ 13-26	10 NP
imul r32,m32,imm	2+d(0-2)+i(1,2)			-	12-41	13-42	10 NP
imul rl6, r16	2+i(1,2)	-		-	9-22	13-18/ 13-26	10 NP
imul r32, r32	2+i(1,2)	-	-		9-38	13-42	10 NP
imul rl6, ml6	2+d(0-2)+i(1,2)	-	-		12-25	13-18/ 13-26	10 NP
imul r32, m32	2+d(02)+i(1,2)				12-41	13-42	10 NP

tutte le forme: dest, org
o
dest, orgl, org2

cicli per: byte/word

IN	**INput from port** **byte**	**8088**	**186**	**286**	**386**	**486**	**Pentium**
in al, imm8	2	14	10	5	12	14	7 NP
in ax, imm8	2	14	10	5	12	14	7 NP
in eax, imm8	2				12	14	7 NP
in al, dx	1	12	8	5	13	14	7 NP
in ax, dx	1	12	8	5	13	14	7 NP
in eax, dx	1				13	14	7 NP

	Modalità protetta **byte**	**386**	**486**	**Pentium**
in acc, imm	2	6/26/26	9/29/27	4/21/19 NP
in acc, dx	1	7/27/27	8/28/27	4/21/19 NP

cicli per: CPL<=IOPL / CPL>IOPL / V86

INC	**INCrement** **byte**	**8088**	**186**	**286**	**386**	**486**	**Pentium**
inc r8	2	3	3	2	2	1	1 UV
inc rl6	1	3	3	2	2	1	1 UV
inc r32	1	3	3	2	2	1	1 UV
inc mem	2+d(0,2)	23+EA	15	7	6	3	3 UV

INS/INSB/INSW/INSD	INput from port to String; input di byte dalla porta DX in ES:DI						
	byte	**8088**	**186**	**286**	**386**	**486**	**Pentium**
insb	1		14	5	15	17	9 NP
insw	1		14	5	15	17	9 NP
insd	1				15	17	9 NP
	Modalità protetta						
	byte				**386**	**486**	**Pentium**
ins	1				9/29/29	10/32/30	6/24/22 NP
	cicli per: CPL <= IOPL / CPL > IOPL / V86						

INT	call INTerrupt procedure						
	byte	**8088**	**186**	**286**	**386**	**486**	**Pentium**
int 3	1	72	45	23+m	33	26	13 NP
int imm8	2	71	47	23+m	37	30	16 NP
	Modalità protetta						
	byte	**8088**	**186**	**286**	**386**	**486**	**Pentium**
int 1	(40-78)+m				59-99	44-71	27-82 NP

INTO	call INTerrupt procedure if Overflow						
	byte	**8088**	**186**	**286**	**386**	**486**	**Pentium**
into	1	4/73	4/48	3/24+m	3/35	3/28	4/13 NP
	Modalità protetta						
	byte			**286**	**386**	**486**	**Pentium**
into	1			(40-78)+m	59-99	44-71	27-56 NP
	Non sono indicati i cicli per !a commutazione del task						

INVD	INValiDate cache (486+)						
	byte	**8088**	**186**	**286**	**386**	**486**	**Pentium**
invd	2					4	15 NP

INVLPG	Invalidate TLB entry (486+)						
	byte					**486**	**Pentium**
invlpg mem32					5	12	25 NP

IRET	Interrupt RETurn						
	byte	**8088**	**186**	**286**	**386**	**486**	**Pentium**
iret	1	44	28	17+m	22	15	8-27 NP
	Non sono indicati i cicli per la commutazione del task						

IRETD	**Interrupt RETurn 32-bit (386+)**			
	byte	**386**	**486**	**Pentium**
iretd	1	22	15	10-27 NP
	Non sono indicati i cicli per la commutazione del task			

JCC	**Jump on Condition Code**						
	byte	**8088**	**186**	**286**	**386**	**486**	**Pentium**
jcc near8	2		4/16	4/13	3/7+m	3/7+m	1/3 1 PV
jcc nearl6	3					3/7+m	1/3 1 PV

cicli per: nessun salto/salto

istruzioni di salto condizionale

ja	jump if above	jnbe	jump if not below or equal
jae	jump if above or equal	jnb	jump if not below
jb	jump if below	jnae	jump if not above or equal
jbe	jump if below or equal	jna	jump if not above
jg	jump if greater	jnle	jump if not less or equal
jge	jump if greater or equal	jnl	jump if not less
j1	jump if less	jnge	jump if not greater or equal
jle	jump if less or equal	jng	jump if not greater
je	jump if equal	jz	jump if zero
jne	jump if not equal	jnz	jump if not zero
jc	jump if carry	jnc	jump if not carry
js	jump if sign	jns	jump if not sign
jnp	jump if no parity (odd)	jpo	jump if parity odd
jo	jump if overflow	jno	jump if not overflow
jp	jump if parity (even)	jpe	jump if parity even

JCXZ/JECXZ	**Jump if CX/ECX=0**						
	byte	**8088**	**186**	**286**	**386**	**486**	**Pentium**
jcxz dest	2	6/18	5/16	4/8+m	5/9+m	5/8	5/6 NP
jecxz dest	2				5/9+m	5/8	5/6 NP
	cicli per: nessun salto/salto						

JMP	**unconditional JuMP**						
	byte	**8088**	**186**	**286**	**386**	**486**	**Pentium**
jmp short	2	15	13	7+m	7+m	3	1 PV
jmp near	3	15	13	7+m	7+m	3	1 PV
jmp far	5	15	13	11+m	12+m	17	3 NP
jmp r16	2	11	11	7+m	7+m	5	2 NP
jmp meml6	2+d(0,2)	18+EA	17	11+m	10+m	5	2 NP
jmp mem32	2+d(4)	24+EA	26	l5+m	12+m	13	4 NP
jmp r32	2	-		-	7+m	5	2 NP
jmp mem32	2+d(0,2)	-	-	-	10+m	5	2 NP
jmp mem48	2+d(6)				12+m	13	4 NP
	non sono indicati i cicli per salti attraverso le porte						

LAHF	**Load in AH the Flags**						
	byte	**8088**	**186**	**286**	**386**	**486**	**Pentium**
lahf	1	4	2	2	2	3	2 NP

LAR	Load Access Rights byte (286+) byte			286	386	486	Pentium
lar rl6, rl6	3			14	15	11	8 NP
lar r32, r32	3			-	15	11	8 NP
lar rl6, ml6	3			16	16	11	8 NP
lar r32, m32	3			-	16	11	8 NP

LDS	Load far pointer byte	8088	186	286	386	486	Pentium
lds reg, mem	2+d(2)	24+EA	18	7	7	6	4 NP

LES	Load far pointer byte	8088	186	286	386	486	Pentium
les reg, mem	2+d(2)	24+EA	18	7	7	6	4 NP

LFS	Load far pointer (386+) byte	386	486	Pentium
lfs reg, mem	3+d(2,4)	7	6	4 NP

LGS	Load far pointer (386+) byte	386	486	Pentium
lgs reg, mem	3+d(2,4)	7	6	4 NP

LSS	Load Stack Segment and offset byte	386	486	Pentium
lss reg, mem	3+d(2,4)	7	6	4 NP

LEA	Load effective address byte	8088	186	286	386	486	Pentium
lea rl6, mem	2+d(2)	2+EA	6	3	2	1-2	1 UV
lea r32, mem	2+d(2)	-	-	-	2	1-2	1 UV

LEAVE	LEAVE high level procedure (186+) byte	186	286	386	486	Pentium
leave	1	8	5	4	5	3 NP

LGDT	Load Global Descriptor Table register (286+) byte	286	386	486	Pentium
lgdt mem48	5	11	11	11	6 NP

LIDT	Load Interrupt Descriptor Table register (286+) byte	286	386	486	Pentium
lidt mem48	5	12	11	11	6 NP

LLDT	**Load Local Descriptor Table register (286+)**				
	byte	**286**	**386**	**486**	**Pentium**
lldt rl6	3	17	20	11	9 NP
lldt meml6	3+d(0-2)	19	24	11	9 NP

LMSW	**Load Machine Status Word (286+)**				
	byte	**286**	**386**	**486**	**Pentium**
lmsw rl6	3	3	10	13	8 NP
lmsw mem16	3+d(0-2)	6	13	13	8 NP

LOCK	**LOCK bus on next instruction (prefisso)**						
	byte	**8088**	**186**	**286**	**386**	**486**	**Pentium**
lock codice-op	1	2	2	0	0	1	1 NP

LODS/LODSB/LODSW/ LODSD		**LOaD String operand**					
	byte	**8088**	**186**	**286**	**386**	**486**	**Pentium**
lodsb	1	16	10	5	5	5	2 NP
lodsw	1	16	10	5	5	5	2 NP
lodsd	1	-	-	-	5	5	2 NP

LOOP	**LOOP control with CX counter**						
	byte	**8088**	**186**	**286**	**386**	**486**	**Pentium**
loop short	2	5/17	5/15	4/8+m	11+m	6/7	5/6 NP
loopw short	(usa CX in modalità a 32 bit)						
loopd short	(usa ECX in modalità a 16 bit)						

LOOPE	**LOOP while Equal**						
	byte	**8088**	**186**	**286**	**386**	**486**	**Pentium**
loope short	2	6/18	5/16	4/8	11+m	6/9	7/8 NP
loopz short							
loopew short	(usa CX in modalità a 32 bit)						
loopzw short	(usa CX in modalità a 32 bit)						
looped short	(usa ECX in modalità a 16 bit)						
loopzd short	(usa ECX in modalità a 16 bit)						

LOOPZ	**LOOP while Zero**
Vedere:	loope (loope e loopz sono sinonimi)

LOOPNE	**LOOP while Not Equal**						
	byte	**8088**	**186**	**286**	**386**	**486**	**Pentium**
loopne short	2	5/19	5/16	4/8	11+m	6/9	7/8 NP
loopnz short							
loopnew short	(usa CX in modalità a 32 bit)						
loopnzw short	(usa CX in modalità a 32 bit)						
loopned short	(usa ECX in modalità a 16 bit)						
loopnzd short	(usa ECX in modalità a 16 bit)						
LOOPNZ	LOOP while Not Zero						
Vedere:	loopne (loopne e loopnz sono sinonimi)						

LSL	**Load Segment Limit (286+)** **byte**	**286**	**386**	**486**	**Pentium**
lsl rl6, r16	3	14	20/25	10	8 NP
lsl r32, r32	3	-	20/25	10	8
lsl rl6, ml6	3+d(0,2)	16	21/26	10	8
lsl r32, m32	3+d(0,2)	-	21/26	10	8

LTR	Load Task Register (286+) byte	286	386	486	Pentium
ltr rl6	3	17	23	20	10 NP
ltr meml6	3+d(0,2)	19	27	20	10

MOV	**MOVe data** **byte**	**8088**	**186**	**286**	**386**	**486**	**Pentium**
mov reg, reg	2	2	2	2	2	1	1 UV
mov mem, reg	2+d(0-2)	13+EA	9	3	2	1	1 UV
mov reg, mem	2+d(0-2)	12+EA	12	5	4	1	1 UV
mov mem, imm	2+d(0-2)+i(1,2)	14+EA	12-13	3	2	1	1 UV*
mov reg, imm	2+i(l,2)	4	34	2	2	1	1 UV
mov acc, mem	3	14	8	5	4	1	1 UV
mov mem, acc	3	14	9	3	2	1	1 UV

* = mov mem+scostamento non è accoppiabile a meno che scostamento = 0

Spostamenti dai registri di segmento

	Modalità reale **byte**	**8088**	**186**	**286**	**386**	**486**	**Pentium**
mov seg, rl6	2	2	2	2	2	3	2-11 NP
mov seg, ml6	2+d(02)	12+EA	9	5	5	3	3-12 NP
mov r16, seg	2	2	2	2	2	3	1 NP
mov ml6, seg	2+d(0,2)	13+EA	11	3	2	3	1 NP

	Differenze per la modalità protetta **byte**	**286**	**386**	**486**	**Pentium**
mov seg, rl6	2	17	18	9	2/11* NP
mov seg, ml6	2+d(0,2)	19	19	9	3/12* NP

* = aggiungere 8 se è un nuovo descrittore, 6 se SS

	MOVE da/verso i registri speciali (386+) **byte**	**386**	**486**	**Pentium**
mov r32, cr32	3	6	4	4 NP
mov cr32, r32	3	4/10*	4/16*	12/22* NP
mov r32, dr32	3	14/22*	10	2/12* NP
mov dr32, r32	3	16/22*	11	11/12* NP
mov r32, tr32	3	12	3/4*	-
mov tr32, r32	3	12	4/6*	-

* = i cicli dipendono dal registro speciale utilizzato.

MOVS/MOVSB/MOVSW/MOVSD — MOVe data from String to string

	byte	8088	186	286	386	486	Pentium
movsb	1	18	9	5	7	7	4 NP
movsw	1	26	9	5	7	7	4 NP
movsd	1	-	-	-	7	7	4 NP
rep movsb	2	9+17n	8+8n	5+4n	7+4n	12+3n*	3+n NP
rep movsw	2	9+25n	8+8n	5+4n	7+4n	12+3n*	3+n NP
rep movsd	2	-	-	-	7+4n	12+3n*	3+n NP

* = 5 se n=0, 13 se n=1 altrimenti 12+3n
(n = numero di byte, word o dword)

MOVSX — MOVe with Sign-eXtend (386+)

	byte	386	486	Pentium
movsx reg, reg	3	3	3	3 NP
movsx reg, mem	3+d(0,1,2,4)	6	3	3 NP

MOVZX — MOVe with Zero-eXtend (386+)

	byte	386	486	Pentium
movzx reg, reg	3	3	3	3 NP
movzx reg, mem	3+d(0,1,2,4)	6	3	3 NP

MUL — unsigned MULtiply

	byte	8088	186	286	386	486	Pentium
mul r8	2	70-77	26-28	13	9-14	13-18	11 NP
mul rl6	2	118-133	35-37	21	9-22	13-26	11 NP
mul r32	2				9-38	1342	10 NP
mul mem8	2+d(0-2)	76-83+EA	32-34	16	12-17	13-18	11 NP
mul mem16	2+d(0-2)	124-139+EA		41-43	24	12-25	13-26
11 NP							
mul mem32	2+d(0-2)	-	-	-	1241	1342	10 NP

moltiplicando implicito		operando (moltiplicatore)		risultato
AL	X	byte	=	AX
AX	X	word	=	DX:AX
EAX	X	dword	=	EDX:EAX

NEG — twos complement NEGation

	byte	8088	186	286	386	486	Pentium
neg reg	2	3	3	2	2	1	1 NP
neg mem	2+d(0-2)	24+EA	13	7	6	3	3 NP

NOP — No OPeration

	byte	8088	186	286	386	486	Pentium
nop	1	3	3	3	3	1	1 UV

NOT — ones complement NOT

	byte	8088	186	286	386	486	Pentium
not reg	2	3	3	2	2	1	1 NP
not mem	2+d(0-2)	24+EA	13	7	6	3	3 NP

OR — **Logical inclusive OR**

	byte	8088	186	286	386	486	Pentium
or reg, reg	2	3	3	2	2	1	1 UV
or mem, reg	2+d(0,2)	24+EA	10	7	7	3	3 UV
or reg, mem	2+d(0,2)	13+EA	10	7	6	2	2 UV
or reg, imm	2+d(1,2)	4	4	3	2	1	1 UV
or mem, imm	2+d(0,2)+i(1,2)	23+EA	16	7	7	3	3 UV*
or acc, imm	1+1 (1,2)	4	4	3	2	1	1 UV

* = non accoppiabile se vi è uno scostamento e un immediato

OUT — **OUTput to port**

	byte	8088	186	286	386	486	Pentium
out imm8, al	2	14	9	3	10	16	12 NP
out imm8, ax	2	14	9	3	10	16	12 NP
out imm8, eax	2	-	-	-	10	16	12 NP
out dx, al	1	12	7	3	11	16	12 NP
out dx, ax	1	12	7	3	11	16	12 NP
out dx, eax	1	-	-	-	11	16	12 NP

Modalità protetta

	byte	386	486	Pentium
out imm8, acc	2	4/24/24	11/31/29	9/26/24 NP
out dx, acc	1	5/25/25	10/30/29	9/26/24 NP

cicli per: CPL <= IOPL / CPL > IOPL / V86

OUTS/OUTSB/OUTSW/OUTSD — **OUTput String to port**

	byte	186	286	386	486	Pentium
outsb	1	14	5	14	17	13 NP
outsw	1	14	5	14	17	13 NP
outsd	1	-	-	14	17	13 NP

Modalità protetta

	byte	386	486	Pentium
outs	1	8/28/28	10/32/30	10/27/25 NP

cicli per: CPL <= IOPL / CPL > IOPL / V86

POP — **POP a word/dword from the stack**

	byte	8088	186	286	386	486	Pentium
pop reg	1	12	10	5	4	1	1 UV
pop mem	2+d(0-2)	25+EA	20	5	5	6	3 NP
pop seg	1	12	8	5	7	3	3 NP
pop FS/GS	2	-	-	-	7	3	3 NP

Modalità protetta

	byte	286	386	486	Pentium
pop CS/DS/ES	1	20	21	9	3-12 NP
pop SS	1	20	21	9	8-17 NP
pop FS/GS	2	-	21	9	3-12 NP

POPA/POPAD — **POP All (186+)/POP All Double (386+)**

	byte	186	286	386	486	Pentium
popa	1	51	19	24	9	5 NP
popad	1	-	-	24	9	5 NP

popa = pop di, si, bp, sp, bx, dx, cx, ax
popad = pop edi, esi, ebp, esp, ebx, edx, ccx, eax
(sp ed esp vengono eliminati)

POPF/POPFD — **POP Flags/POP Flags Double (386+)**

	byte	8088	186	286	386	486	Pentium
popf	1	12	8	5	5	9	6 NP
popfd	1	-	-	-	5	9	6 NP

Modalità protetta

	byte	286	386	486	Pentium
popf	1	5	5	6	4 NP
popfd	1	-	5	6	4 NP

PUSH — **PUSH a word/dword to the stack**

	byte	8088	186	286	386	486	Pentium
push reg	1	15	10	3	21	1 UV	
push mem	2+d(0-2)	24+EA	16	5	5	4	2 NP
push seg	1	14	9	3	2	3	1 NP
push imm	1+i(1,2)	-	-	3	2	1	1 NP
push FS/GS	2	-	-	-	2	3	1 NP

PUSHA/PUSHAD — **PUSH All (186+)/PUSH All Double (386+)**

	byte	186	286	386	486	Pentium
pusha	1	36	17	18	11	5 NP
pushad	1	-	-	18	11	5 NP

pusha = push ax, cx, dx, bx, sp, bp, si, di,
pushad = push eax, ecx, edx, ebx, esp, ebp, esi, edi

PUSHF/PUSHFD — **PUSH Flags/PUSH Flags Double (386+)**

	byte	8088	186	286	386	486	Pentium
pushf	1	14	9	3	4	4	9 NP
pushfd	1	-	-	-	4	4	9 NP

Modalità protetta

	byte	286	386	486	Pentium
pushf	1	3	4	3	3 NP
pushfd	1	-	4	3	3 NP

RCL	**Rotate bits Left with CF**						
	byte	**8088**	**186**	**286**	**386**	**486**	**Pentium**
rcl reg, 1	2	2	2	2	9	3	1 PU
rcl mem, 1	2+d(0,2)	23+EA	15	7	10	4	3 PU
rcl reg, cl	2	8+4n	5+n	5+n	9	8-30	7-24 NP
rcl mem, cl	2+d(0,2)	28+EA+4n	17+n	8+n	10	9-3	9-26
rcl reg, imm	3	-	5+n	5+n	9	8-30	8-25 NP
rcl mem, imm	3+ d(0,2)	-	17+n	8+n	10	9-31	10-27 NP

RCR	**Rotate bits Right with CF**						
	byte	**8088**	**186**	**286**	**386**	**486**	**Pentium**
rcr reg, 1	2	2	2	2	9	3	1 PU
rcr mem, 1	2+d(0,2)	23+EA	15	7	10	4	3 PU
rcr reg, cl	2	8+4n	5+n	5+n	9	8-30	7-24 NP
rcr mem, cl	2+d(0,2)	28+EA+4n	17+n	8+n	10	9-31	9-26 NP
rcr reg, imm	3	-	5+n	5+n	9	8-30	8-25 NP
rcr mem, imm	3+d(0,2)	-	17+n	8+n	10	9-31	10-27 NP

ROL	**ROtate bits Left**						
	byte	**8088**	**186**	**286**	**386**	**486**	**Pentium**
rol reg, 1	2	2	2	2	3	3	1 PU
rol mem, 1	2+d(0,2)	23+EA	15	7	7	4	3 PU
rol reg, cl	2	8+4n	5+n	5+n	3	3	4 NP
rol mem, cl	2+d (0,2)	28+EA+4n	17+n	8+n	7	4	4 NP
rol reg, imm	3	-	5+n	5+n	3	2	1 PU
rol mem, imm	3+d(0,2)	-	17+n	8+n	7	4	3 PU*
	* = non accoppiabili se vi è uno scostamento e una costante						

ROR	**ROtate bits Right**						
	byte	**8088**	**186**	**286**	**386**	**486**	**Pentium**
ror reg, 1	2	2	2	2	3	3	1 PU
ror mem, 1	2+d(0,2)	23+EA	15	7	7	4	3 PU
ror reg, cl	2	8+4n	5+n	5+n	3	3	4 NP
ror mem, c1	2+d(0,2)	28+EA+4n	17+n	8+n	7	4	4 NP
ror reg, imm	3	-	5+n	5+n	3	2	1 PU
ror mem, imm	3+d(0,2)	17+n	8+n	7	4	3	PU*
	* = non accoppiabili se vi è uno scostamento e una costante						

RDMSR	**ReaD from Model-Specific Register (Pentium)**	
	byte	**Pentium**
rdmsr	2	2-24 NP

REP	**REPeat string operation**
Vedere:	rep movs
Vedere:	rep stos

REPE	**REPeat while Equal (or zero) string operation**	
Vedere:	repe cmps	trova elementi in memoria non corrispondenti
Vedere:	repe scas	trova byte in memoria non corrispondenti

REPNE — REPeat while Not Equal (or not zero) string operation

Vedere:	repne cmps	trova i primi elementi in memoria corrispondenti
Vedere:	repne scas	trova il primo elemento in memoria corrispondente con acc

RET/RETN/RETF — RETurn from procedure

Per l'istruzione RET l'assembler produce RETN (return near) o RETF (return far)

	byte	8088	186	286	386	486	Pentium
retn	1	20	16	11+m	10+m	5	2 NP
retn imml6	1+d(2)	24	18	11+m	10+m	5	3 NP
retf	1	34	22	15+m	18+m	13	4 NP
ret inl6	1+d(2)	33	25	15+m	18+m	14	4 NP

Modalità protetta

	byte	286	386	486	Pentium
retf	1	25+m/55	32+m/62	18/33	4-13/23 NP
retf inl6	1+d(2)	25+m/55	32+m/68	17/33	4-13/23 NP

cicli per: stesso livello di privilegio/livello di privilegio inferiore

RSM — Resume from System Management mode (Pentium)

	byte	Pentium
rsm	2	83 NP

SAL/SHL/SAR/SHR — SHift bits

	byte	8088	186	286	386	486	Pentium
sh reg, 1	2	2	2	2	3	3	1 PU
sh mem, 1	2+d(0,2)	23+EA	15	7	7	4	3 PU
sh reg, cl	2	8+4n	5+n	5+n	3	3	4 NP
sh mem, cl	2+d(0,2)	28+EA+4n	17+n	8+n	7	4	4 NP
sh reg, imm	3	-	5+n	5+n	3	2	1 PU
sh mem, imm	3+d(0,2)	-	17+n	8+o	7	4	3 PU*

* = non accoppiabili se vi è uno scostamento e una costante

sh= sal, shl, sar o shr

sal = Shift Arithmetic Left — sar = Shift Arithmetic Right

shl = SHift Left (come sal) — shr = SHift Right

SAHF — Store AH into Flags

	byte	8088	186	286	386	486	Pentium
sahf	1	4	3	2	3	2	2 NP

SBB — Integer SuBtraction with Borrow

	byte	8088	186	286	386	486	Pentium
sbb reg, reg	2	3	3	2	2	1	1 PU
sbb mem, reg	2+d(0,2)	24+EA	10	7	7	3	3 PU
sbb reg, mem	2+d(0,2)	13+EA	10	7	6	2	2 PU
sbb reg, imm	2+d(1,2)	4	4	3	2	1	1 PU
sbb mem, imm	2+d(0,2)+i(1,2)	23+EA	16	7	7	3	3 PU*
sbb acc, imm	1+i(1,2)	4	4	3	2	1	1 PU

* = non accoppiabili se vi è uno scostamento e una costante

SCAS/SCASB/SCASW/SCASD		SCAn String data					
	byte	8088	186	286	386	486	Pentium
scasb	1	19	15	7	7	6	4 NP
scasw	1	19	15	7	7	6	4 NP
scasd	1	-	-	-	7	6	4 NP
repX scasb	2	9+15n	5+15n	5+8n	5+8n	7+5n*	8+4n NP
repX scasw	2	9+19n	5+15n	5+8n	5+8n	7+5n*	8+4n NP
repX scasd	2				5+8n	7+5n*	8+4n NP

repX = repe, repz, repne o repnz
* se n=0 (n = numero di byte, word o dword)

SET	SET byte to 1 on condition else set to 0 (386+)			
	byte	386	486	Pentium
setCC reg	3	4	4/3	1/2 NP
setCC mem	3+ d(0-2)	5	3/4	1/2 NP

Cicli per: true/false
setCC può essere :

seta	setae	setb	setbe	setc
setg	setge	setl	setle	setna
setnb	setnbe	setnc	setne	setng
setnl	setnle	setno	setnp	setns
seto	setp	setpe	setpo	sets

SGDT	Store Global Descriptor Table register (286+)				
	byte	286	386	486	Pentium
sgdt mem48	5	11	9	10	4 NP

SIDT	Store Interrupt Descriptor Table register (286+)				
	byte	286	386	486	Pentium
sidt mem48	5	12	9	10	4 NP

SHLD	Double precision SHift Left (386+)			
	byte	386	486	Pentium
shld reg, reg, imm	4	3	2	4 NP
shld mem, reg, imm	4+d(0-2)	7	3	4 NP
shld reg, reg, cl	4	3	3	4 NP
shld mem, reg, cl	4+d(0-2)	7	4	5 NP

SHRD	Double precision SHift Right (386+)			
	byte	386	486	Pentium
shrd reg, reg, imm	4	3	2	4 NP
shrd mem, reg, imm	4+ d(0-2)	7	3	4 NP
shrd reg, reg, cl	4	3	3	4 NP
shrd mem, reg, cl	4+ d(0-2)	7	4	5 NP

SLDT	Store Local Descriptor Table register (286+)				
	byte	286	386	486	Pentium
sldt reg	3	2	2	2	2 NP
sldt mem	3+d(0-2)	3	2	3	2 NP

SMSW	**Store Machine Status Word (286+)**					
	byte	**286**	**386**	**486**	**Pentium**
smsw reg	3	2	2	2	4 NP
smsw mem	3+d(0-2)	3	3	3	4 NP

STC	**SeT the Carry flag**						
	byte	**8088**	**186**	**286**	**386**	**486**	**Pentium**
stc	1	2	2	2	2	2	2 NP

STD	**SeT Direction flag (all'indietro)**						
	byte	**8088**	**186**	**286**	**386**	**486**	**Pentium**
std	1	2	2	2	2	2	2 NP

STI	**SeT Interrupt flag (enable)**						
	byte	**8088**	**186**	**286**	**386**	**486**	**Pentium**
sti	1	2	2	2	3	5	7 NP

STOS/STOSB/STOSW/STOSD		**STOre String data**					
	byte	**8088**	**186**	**286**	**386**	**486**	**Pentium**
stosb	1	11	10	3	4	5	3 NP
stosw	1	15	10	3	4	5	3 NP
stosd	1	-	-	-	4	5	3 NP
rep stosb	2	9+10n	6+9n	4+3n	5+5n	7+4n*	3+n NP
rep stosw	2	9+14n	6+9n	4+3n	5+5n	7+4n*	3+n NP
rep stosd	2	-	-	-	5+5n	7+4n*	3+n NP

* = 5 se n=0, 13 se n=1 (n = numero di byte, word o dword)

STR	**Store Task Register (286+)**				
	byte	**286**	**386**	**486**	**Pentium**
str reg	3	2	2	2	2 NP
str mem	3+d(0-2)	3	2	3	2 NP

SUB	**integer SUBtraction**						
	byte	**8088**	**186**	**286**	**386**	**486**	**Pentium**
sub reg, reg	2	3	3	2	2	1	1 UV
sub mem, reg	2+d(0,2)	24+EA	10	7	7	3	3 UV
sub reg, mem	2+d(0,2)	13+EA	10	7	6	2	2 UV
sub reg, imm	2+d(1,2)						
sub mem, imm	2+d(0,2)	4	4	3	2	1	1 UV
	+i(1,2)	23+EA	16	7	7	3	3 UV
sub acc, imm	1+1 (1,2)	4	4	3	2	1	1 UV

* = non accoppiabili se vi è uno scostamento e una costante

TEST	logical TEST byte	8088	186	286	386	486	Pentium
test reg, reg	2	3	3	2	2	1	1 UV
test mem, reg	2+d(0,2)	13+EA	10	6	5	2	2 UV
test reg, mem	2+d(0,2)	13+EA	10	6	5	2	2 UV
test reg, imm	2+d(1,2)						
test mem, imm	2+d(0,2)	5	4	3	2	1	1 UV
	+1 (1,2)	11+EA	10	6	5	2	2 UV*
test acc, imm	1+1 (1,2)	4	4	3	2	1	1 UV
	* = non accoppiabili se vi è uno scostamento e una costante						

VERR	VERify a segment for Reading (286+) byte	286	386	486	Pentium
verr reg	3	14	10	11	7 NP
verr mem	3+d(0,2)	16	11	11	7 NP

VERW	Verify a segment for Writing (286+) byte	286	386	486	Pentium
verw reg	3	14	15	11	7 NP
verw mem	3+d(0,2)	16	16	11	7 NP

WAIT	WAIT for co-processor byte	8088	186	286	386	486	Pentium
wait	1	4	6	3	6	1-3	1 NP

WBINVD	Write-Back and INValidate Data cache (486+) byte	486	Pentium
wbinvd	2	5	2000+ NP

WRMSR	WRite to Model-Specific Register (PENTIUM) byte	Pentium
wrmsr	2	30-5 NP

XADD	eXchange and ADD (486+) byte	486	Pentium
xadd reg, reg	3	3	3 NP
xadd mem, reg	3+d(0-2)	4	4 NP

XCHG	eXCHanGe register/memory with register byte	8088	186	286	386	486	Pentium
xchg reg, reg	2	4	4	3	3	3	3 NP
xchg reg, mem	2+d(0-2)	25+EA	17	5	5	5	3 NP
xchg mem, reg	2+d(0-2)	25+EA	17	5	5	5	3 NP
xchg acc, reg	1	3	3	3	3	3	2 NP
xchg reg, acc	1	3	3	3	3	3	2 NP
	acc = solo AX o LAX						

XLAT/XLATB	**TRANSLATion table look-up**						
	byte	**8088**	**186**	**286**	**386**	**486**	**Pentium**
xlat		1		11 11	5	5 4	4 NP
xlatb							

XOR	**logical eXclusive OR**						
	byte	**8088**	**186**	**286**	**386**	**486**	**Pentium**
xor reg, reg	2	3	3	2	2	1	1 UV
xor mem, reg	2+d(0,2)	24+EA	10	7	7	3	3 UV
xor reg, mem	2+d(0,2)	13+EA	10	7	6	2	2 UV
xor reg, imm	2+d(1,2)	4	4	3	2	1	1 UV
xor mem, imm	2+d(0,2)						
	+i(1,2)	23+EA	16	7	7	3	3 UV
xor acc, imm	1+i(1,2)	4	4	3	2	1	1 UV
	* = non accoppiabili se vi è uno scostamento e una costante						

Note :

acc = AL, AX o EAX se non viene specificato diversamente
reg = qualsiasi registro generale
r8 = qualsiasi registro generale a 8 bit
r16 = qualsiasi registro generale a 16 bit
r32 = qualsiasi registro generale a 32 bit
imm = valori immediati (costanti)
imm8 = valori immediati a 8 bit
imml6 = valori immediati a 16 bit
mem = indirizzo di memoria
mem8 = indirizzo di memoria a 8 bit
meml6 = indirizzo di memoria a 16 bit
mem32 = indirizzo di memoria a 32 bit

n = generalmente fa riferimento a valori ripetuti
m = in un salto o in una chiamata
286: byte nella prossima istruzione
386/486: numero di componenti
(ogni byte del codice operativo) + 1 (se vi sono dati immediati) + 1 (se vi è uno scostamento)

Lunghezza delle istruzioni

I byte indicati includono la lunghezza del codice operativo (opcode) e di tutti i dati (scostamenti, dati immediati e così via) richiesti. Se lo scostamento è opzionale, viene indicato come *d*() e le relative lunghezze vengono indicate fra parentesi. Se i dati immediati (costanti) sono opzionali, vengono indicati come *i*()e le relative lunghezze vengono indicate fra parentesi.

Accoppiabilità delle istruzioni sul Pentium

NP = non accoppiabile
UV = accoppiabile sulle pipe U o V
PU = accoppiabile solo su U
PV = accoppiabile solo su V

A.2 Set di istruzioni 80x87 (8087 - Pentium)

Questa sezione include i tempi di esecuzione di tutte le istruzioni in virgola mobile e informazioni specifiche di accoppiamento di istruzioni per il Pentium. Per informazioni sulle abbreviazioni utilizzate, si consulti la legenda posta al termine di questa appendice.

F2XM1	**Floating-point 2x-1**				
	8087	**287**	**387**	**486**	**Pentium**
f2xml	310-630	310-630	211-476	140-279	13-57 NP

FABS	**Floating-point ABSolute value**				
	8087	**287**	**387**	**486**	**Pentium**
fabs	10-17	10-17	22	3	1 FX

FADD/FADDP	**Floating point ADD/FADDP**			**Floating point ADD and Pop**	
	8087	**287**	**387**	**486**	**Pentium**
fadd	70-100	70-100	23-34	8-20	3/1 FX
fadd mem32	90-120+EA	90-120	24-32	8-20	3/1 FX
fadd mem64	95-125+EA	95-125	29-37	8-20	3/1 FX
faddp	75-105	75-105	23-31	8-20	3/1 FX

FBLD	**Floating-point BcD LoaD**				
	8087	**287**	**387**	**486**	**Pentium**
fbld mem	(290-310)+EA	290-310	266275	70-103	48-58 NP

FBSTP	**Floating-point Bcd STore and Pop**				
	8087	**287**	**387**	**486**	**Pentium**
fbstp	(520-540)+PA	520-540	512-534	172-176	148-154 NP

FCHS	**Floating-point CHange Sign**				
	8087	**287**	**387**	**486**	**Pentium**
fchs	10-17	10-17	24-25	6	1 FX

FCLEX **FNCLEX**	**Floating-point CLear EXceptions** **Floating-point No wait and CLear EXceptions** **8087**	**287**	**387**	**486**	**Pentium**
fclex	2-8	2-8	11	7	9 NP
fnclex	2-8	2-8	11	7	9 NP
	La prima versione può richiedere cicli aggiuntivi				

FCOM **FCOMP** **FCOMPP**	**Floating-point COMpare** **Floating-point COMpare and Pop** **Floating-point COMpare and Pop twice** **8087**	**287**	**387**	**486**	**Pentium**
fcom reg	40-50	40-50	24	4	4/1 FX
fcom mem32	(60-70)+EA	60-70	26	4	4/1 FX
fcom mem64	(65-75)+li/t	65-75	31	4	4/1 FX
fcomp	42-52	42-52	26	4	4/1 FX
fcompp	45-55	45-55	26	5	4/1 FX

FCOS	**Floating-point COSine (387+)** **8087**	**287**	**387**	**486**	**Pentium**
fcos	-	-	123-772	257-354	18-124 NP
	Ulteriori cicli se operando > p/4				

FDECSTP	**Floating-point DECrement STack Pointer** **8087**	**287**	**387**	**486**	**Pentium**
fdecstp	612	612	22	3	1 NP

FDISI **FNDISI**	**Floating-point DISable Interrupts (solo 8087, gli altri eseguono l'istruzione fnop)** **Floating-point No wait DISable Interrupts (solo 8087, gli altri eseguono l'istruzione fnop)** **8087**	**287**	**387**	**486**	**Pentium**
fdisi	2-8	2	2	3	1 NP
fndisi	2-8	2	2	3	1 NP
	La versione "wait" può richiedere cicli aggiuntivi				

FDIV **EDIVP**	**Floating DIVide** **Floating DIVide and Pop** **8087**	**287**	**387**	**486**	**Pentium**
fdiv reg	193-203	193-203	88-91	73	39 FX
fdiv mem32	(215-225)+EA	215-225	89	73	39 FX
fdiv mem64	(220-230)+EA	220-230	94	73	39 FX
fdivp	197-207	197-207	91	73	39 FX

FDIVR	**Floating DIVide Reversed**				
FDIVRP	**Floating DIVide Reversed and Pop**				
	8087	**287**	**387**	**486**	**Pentium**
fdivr reg	194-204	194204	88-91	73	39 FX
fdivr mem32	(216-226)+EA	216226	89	73	39 FX
fdivr mem64	(221-231)+EA	221-231	94	73	39 FX
fdivrp	198-208	198-208	91	73	39 FX

FENI	**Floating ENable Interrupts (solo 8087, gli altri coprocessori eseguono l'istruzione fnop)**				
FNENI	**Floating No wait ENable Interrupts (solo 8087, gli altri coprocessori eseguono l'istruzione fnop)**				
	8087	**287**	**387**	**486**	**Pentium**
feni	2-8	2	2	3	1 NP
fneni	2-8	2	2	3	1 NP

FFREE	**Floating FREE register**				
	8087	**287**	**387**	**486**	**Pentium**
ffree	9-16	9-16	18	3	1 NP

FIADD	**Floating Integer ADD**				
	8087	**287**	**387**	**486**	**Pentium**
fiadd meml6	(102-137)+EA	102-137	71-85	20-35	7/4 NP
fiadd mem32	(108-143)+EA	108-143	57-72	19-32	7/4 NP

FICOM	**Floating Integer COMpare**				
FICOMP	**Floating Integer COMpare and Pop**				
	8087	**287**	**387**	**486**	**Pentium**
ficom meml6	(72-86)+FA	72-86	71-75	1620	8/4 NP
ficom mem32	(78-91)+EA	78-91	5663	15-17	8/4 NP
ficomp meml6	(74-88)+EA	74-88	71-75	16-20	8/4 NP
ficomp mem32	(80-93)+EA	80-93	5663	15-17	8/4 NP

FIDIV	**Floating Integer DIVide**				
FIDIVR	**Floating Integer DIVide Reversed**				
	8087	**287**	**387**	**486**	**Pentium**
fidiv meml6	(224-238)+EA	224-238	136-140	85-89	42 NP
fidiv mem32	(230-243)+EA	230-243	120-127	84-86	42 NP
fidivr meml6	(225-239)+EA	225-239	135-141	85-89	42 NP
fidivr mem32	(231-245)+PA	231-245	121-128	84-86	42 NP

FILD	**Floating Integer LoaD**				
	8087	**287**	**387**	**486**	**Pentium**
fild meml6	(46-54)+EA	46-54	61-65	13-16	3/1 NP
fild mem32	(52-60)+EA	52-60	45-52	9-12	3/1 NP
fild mem64	(60-68)+EA	60-68	56-67	10-18	3/1 NP

FIMUL	**Floating Integer MULtiply**				
	8087	**287**	**387**	**486**	**Pentium**
fimul meml6	(124-138)+EA	124-138	76-87	23-27	7/4 NP
fimul mem32	(130-144)+EA	130-144	61-82	22-24	7/4 NP

FINCSTP	**Floating INCrement STack Pointer**				
	8087	**287**	**387**	**486**	**Pentium**
fincstp	6-12	6-12	21	3	1 NP

FINIT **FNINIT**	**Floating point processor INITialize** **Floating No-wait point processor INITialize**				
	8087	**287**	**387**	**486**	**Pentium**
finit	2-8	2-8	33	17	16 NP
fninit	2-8	2-8	33	17	12 NP
	La versione “wait” può richiedere cicli aggiuntivi				

FIST **FISTP**	**Floating point Integer STore** **Floating point Integer STore and Pop**				
	8087	**287**	**387**	**486**	**Pentium**
fist meml6	(80-90)+EA	80-90	82-95	29-34	6 NP
fist mem32	(82-92)+EA	82-92	79-93	28-34	6 NP
fistp meml6	(82-92)+EA	82-92	82-95	29-34	6 NP
fistp mem32	(84-94)+EA	84-94	79-93	28-34	6 NP
fistp mem64	(94-105)+EA	94-105	80-97	28-34	6 NP

FISUB **FISUBR**	**Floating-point Integer SUBtract** **Floating-point Integer SUBtract Reversed**				
	8087	**287**	**387**	**486**	**Pentium**
fisub meml6	(102-137)+EA	102-137	71-85	20-35	7/4 NP
fisubr mem32	(108-143)+FA	108-143	57-82	19-32	7/4 NP

FLD	**Floating point LoaD**				
	8087	**287**	**387**	**486**	**Pentium**
fld reg	17-22	17-22	14	4	1 FX
fid mem32	(38-56)+EA	38-56	20	3	1 FX
fld mem64	(40-60)+EA	40-60	25	3	1 FX
fld mem80	(53-65)+EA	53-65	44	6	3 NP

FLD1	**Floating-point LoaD constant onto stack, 1.0**				
FLDL2E	**Floating-point LoaD constant onto stack, Logarithm base 2 (e)**				
FLDL2T	**Floating-point LoaD constant onto stack, Logarithm base 2 (10)**				
FLDLG2	**Floating-point LoaD constant onto stack, Logarithm base 10 (2)**				
FLDLN2	**Floating-point LoaD constant onto stack, Natural logarithm (2)**				
FLDPI	**Floating-point LoaD constant onto stack, PI (3.14159...)**				
	8087	**287**	**387**	**486**	**Pentium**
fldz	11-17	11-17	20	4	2 NP
fldl	15-21	15-21	24	4	2 NP
fldl2e	15-21	15-21	40	8	5/3 NP
fldl2t	16-22	16-22	40	8	5/3 NP
fldlg2	18-24	18-24	41	8	5/3 NP
fldln2	17-23	17-23	41	8	5/3 NP
fldpi	16-22	16-22	40	8	5/3 NP

FLDCW	**Floating-point LoaD Control Word**				
	8087	**287**	**387**	**486**	**Pentium**
fldcw meml6	(7-14)+EA	7-14	19	4	7 NP

FLDENV	**Floating-point LoaD ENVironment state**				
	8087	**287**	**387**	**486**	**Pentium**
fldenv mem	(35-45)+EA	35-45	71	44/34	37/32-33 NP
	cicli per modalità reale/modalità protetta				

FMUL	**Floating-point MULtiply**					
FMULP	**Floating-point MULtiply and Pop**					
	8087	**287**	**387**	**486**	**Pentium**	
fmul reg s	90-105	90-105	29-52	16	3/1	FX
fmul reg	130-145	130-145	46-57	16	3/1	FX
fmul mem32	(110-125)+EA	110-125	27-35	11	3/1	FX
fmul mem64	(154-168)+EA	154-168	32-57	14	3/1	FX
fmulp reg s	94-108	94-108	29-52	16	3/1	FX
fmulp reg	134-148	134-148	29-57	16	3/1	FX
	s = registro con 40 zero iniziali nella frazione					

FNOP	**Floating-point NO OPeration**				
	8087	**287**	**387**	**486**	**Pentium**
fnop	10-16	10-16	12	3	1 NP

FPATAN	**Floating-point Partial ArcTANgent**				
	8087	**287**	**387**	**486**	**Pentium**
fpatan	250-800	250-800	314-487	218-303	17-173

FPREM	**Floating-point Partial REMainder**				
FPREMI	**Floating-point Partial REMainder (Ieee compatible, 387+)**				
	8087	**287**	**387**	**486**	**Pentium**
fprem	15-190	15-190	74-155	70-138	16-64 NP
fpreml	_	_	95-185	72-167	20-70 NP

FPTAN — Floating-point Partial TANgent

	8087	287	387	486	Pentium
fptan	30-540	30-540	191-497	200-273	17-173 NP

Sono necessari cicli aggiuntivi se operando > p/4

FRNDINT — Floating-point RouND to INTeger

	8087	287	387	486	Pentium
frndint	16-50	16-50	66-80	21-30	9-20 NP

FRSTOR — Floating-point ReSTORe saved state

	8087	287	387	486	Pentium
frstor mem	(197-207)+EA	197-207	308	131/120	75-95/70 NP
frstorw mem	-	-	308	131/120	75-95/70 NP
frstord mem	-	-	308	131/120	75-95/70 NP

cicli per modalità reale/modalità protetta

FSAVE — Floating-point SAVE fpu state
FSAVEW — Floating-point SAVE fpu state, Word format (387+)
FSAVED — Floating-point SAVE fpu state, Dword format (387+)
FNSAVE — Floating-point No wait SAVE fpu state
FNSAVEW — Floating-point No wait SAVE fpu state, Word format (387+)
FNSAVED — Floating-point No wait SAVE fpu state, Dword format (387+)

	8087	287	387	486	Pentium
fsave	(197-207)+EA	197-207	375-376	154/143	127-151/124 NP
fsavew	-	-	375-376	154/143	127-151/124 NP
fsaved	-	-	375-376	154/143	127-151/124 NP
fnsave	(197-207)+EA	197-207	375-376	154/143	127-151/124 NP
fnsavew	-	-	375-376	154/143	127-151/124 NP
fnsaved	-	-	375-376	154/143	127-151/124 NP

Cicli per modalità reale/modalità protetta
La versione "wait" può richiedere cicli aggiuntivi

FSCALE — Floating-point SCALE by factor of 2

	8087	287	387	486	Pentium
fscale	32-38	32-38	67-86	30-32	20-31 NP

FSETPM — Floating-point SET Protected Mode (solo 287, 387+ = fnop)

	8087	287	387	486	Pentium
fsetpm	-	2-8	12	3	1 NP

FSIN — Floating-point SINe (387+)
FSINCOS — Floating-point SINe and COSine (387+)

	8087	287	387	486	Pentium
fsin	-	-	122-771	257-354	16-126 NP
fsincos	-	-	194-809	292-365	17-137 NP

Sono necessari cicli aggiuntivi se operando > p/4

FSQRT	**Floating-point SQuare RooT**				
	8087	**287**	**387**	**486**	**Pentium**
fsqrt	180-186	180-186	122-129	83-87	70 NP

FST **FSTP**	**Floating point STore** **Floating point STore and Pop**				
	8087	**287**	**387**	**486**	**Pentium**
fst reg	15-22	15-22	11	3	1 NP
fst mem32	(84-90)+EA	84-90	44	7	2 NP
fst mem64	(96-104)+EA	96-104	45	8	2 NP
fstp reg	17-24	17-24	12	3	1 NP
fstp mem32	(8692)+EA	86-92	44	7	2 NP
fstp mem64	(98-106)+EA	98-106	45	8	2 NP
fstp mem80	(52-58)+EA	52-58	53	6	3 NP

FSTCW **FNSTCW**	**Floating-point STore Control Word** **Floating-point No wait STore Control Word**				
	8087	**287**	**387**	**486**	**Pentium**
fstcw mem	12-18	12-18	15	3	2 NP
fnstcw mem	12-18	12-18	15	3	2 NP
	La versione "wait" può richiedere cicli aggiuntivi				

FSTENV **FSTENVW** **FSTENVD** **FNSTENV** **FNSTENVW** **FNSTENVD**	**Floating-point STore fpu ENVironment** **Floating-point STore fpu ENVironment, Word format (387+)** **Floating-point STore fpu ENVironment, Dword format (387** **Floating-point No-wait STore fpu ENVironment** **Floating-point No-wait STore fpu ENVironment, Word format (387+)** **Floating-point No-wait STore fpu ENVironment, Dword format (387+)**				
	8087	**287**	**387**	**486**	**Pentium**
fstenv mem	(40-50)+EA	40-50	103-104	67/56	48-50 NP
fstenvw mem	-	-	103-104	67/56	48-50 NP
fstenvd mem	-	-	103-104	67/56	48-50 NP
fnstenv mem	(40-50)+EA	40-50	103-104	67/56	48-50 NP
fnstenvw mem	-	-	103-104	67/56	48-50 NP
fnstenvd mem	-	-	103-104	67/56	48-50 NP
	Cicli per modalità reale/modalità protetta La versione "wait" può richiedere cicli aggiuntivi				

FSTSW **FNSTSW**	**Floating-point STore Status Word** **Floating-point No-wait STore Status Word**				
	8087	**287**	**387**	**486**	**Pentium**
fstsw mem	12-18	12-18	15	3	2 NP
fstsw ax	-	10-16	13	3	2 NP
fnstsw mem	12-18	12-18	15	3	2 NP
fnstsw ax	-	10-16	13	3	2 NP
	La versione "wait" può richiedere cicli aggiuntivi				

FSUB **FSUBP**	**Floating-point SUBtract** **Floating-point SUBtract and Pop** **8087**	**287**	**387**	**486**	**Pentium**
fsub reg	70-100	70-100	26-37	8-20	3/1 FX
fsub mem32	(90-120)+EA	90-120	24-32	8-20	3/1 FX
fsub mem64	(95-125)+EA	95-125	28-36	8-20	3/1 FX
fsubp reg	75-105	75-105	26-34	8-20	3/1 FX

FSUBR **FSUBRP**	**Floating-point SUBtract Reversed** **Floating-point SUBtract Reversed and Pop** **8087**	**287**	**387**	**486**	**Pentium**
fsubr reg	70-100	70-100	26-37	8-20	3/1 FX
fsubr mem32	(90-120)+EA	90-120	24-32	8-20	3/1 FX
fsubr mem64	(95-125)+EA	95-125	28-36	8-20	3/1 FX
fsubrp reg	75-105	75-105	2634	8-20	3/1 FX

FTST	**Floating-point TeST for zero** **8087**	**287**	**387**	**486**	**Pentium**
ftst	3848	38-48	28	4	4/1 FX

FUCOM FUCOMP FUCOMPP	Floating-point Unordered COMpare (387+) Floating-point Unordered COMpare and Pop (387+) Floating-point Unordered COMpare and Pop twice (387+) 8087	287	387	486	Pentium
fucom	-	-	24	4	4/1 FX
fucomp	-	-	26	4	4/1 FX
fucompp	-	-	26	5	4/1 FX

FWAIT	**Floating-point WAIT while fpu is executing** **8087**	**287**	**387**	**486**	**Pentium**
fwait	4	3	6	1-3	1-3 NP

FXAM	**Floating-point eXAMine condition flags** **8087**	**287**	**387**	**486**	**Pentium**
fxam	12-23	12-23	30-38	8	21 NP

FXCH	**Floating-point eXCHange floating point registers** **8087**	**287**	**387**	**486**	**Pentium**
fxch	10-15	10-15	18	4	0-1 *
	FXCH è accoppiabile nella pipe V con tutte le istruzioni FX accoppiabili				

FXTRACT	**Floating-point eXTRACT exponent and significand** **8087**	**287**	**387**	**486**	**Pentium**
fxtract	27-55	27-55	70-76	16-20	13 NP

FYL2X FYL2XP1	Floating-point Y * Log2 (X) Floating-point Y * Log2 (X + 1) 8087	287	387	486	Pentium
fyl2x	900-1100	900-1100	120-538	196329	22-111 NP
fyl2xp1	700-1000	700-1000	257-547	171-326	22-103 NP

Tempi di esecuzione delle istruzioni in virgola mobile

FX = accoppiabile con FXCH
NP = non accoppiabile

I valori indicati con un trattino indicano che i tempi cadono in un determinato intervallo. I valori indicati con una barra (quando non è indicato diversamente) corrispondono alla latenza e alla velocità di passaggio delle istruzioni. La latenza fra un'istruzione e la successiva dipende dal risultato. La velocità di passaggio delle istruzioni nella pipeline si considera fra istruzioni che non entrano in conflitto fra loro.

Dimensioni delle istruzioni in virgola mobile

Tutte le istruzioni in virgola mobile che non accedono alla memoria sono lunghe due byte (tranne FWAIT, che è lunga 1 byte).

Le istruzioni in virgola mobile che accedono alla memoria sono lunghe 4 byte per indirizzi a 16 bit e 6 byte per indirizzi a 32 bit.

Sul coprocessore 8087 l'assembler inserisce automaticamente un'istruzione WAIT (FWAIT) prima di ogni istruzione in virgola mobile.

Appendice B

Ottimizzazione delle istruzioni, guida alfabetica

Quello che segue è un elenco di istruzioni e/o sequenze di istruzioni che possono essere ottimizzate (con le relative sostituzioni) quando si ricerca una maggior velocità e/o compattezza del codice.

Legenda:

Istruzioni originali	**CPU/modalità**
Descrizione funzionale o istruzioni sostituibili	Descrizione

aad (imm8)	**tutte le CPU**
AL = AL + (AH * imm8) AH = 0	Se imm8 è vuoto viene usato il valore 10. Molto spesso è più lenta ma è lunga solo 2 byte
aam (imm8)	**tutte le CPU**
AH = AL / imm8 AL = AL MOD imm8	Come AAD.
adc	**Pentium**
	Accoppiabile solo nella pipe U.
add	**modalità a 16 bit della CPU**
lea reg, [reg+reg+sco]	Usare LEA per sommare base + indice + scostamento. Conserva anche i flag.
add	**modalità a 32 bit della CPU**
lea reg, [reg+reg*scala+sco]	Usare LEA per sommare base + indice per la scala + scostamento. Conserva anche i flag.
and reg, reg	**Pentium**
test reg, reg	Meno conflitti di registro e dunque maggiore accoppiabilità con altre istruzioni.

bswap	**Pentium**
ror eax, 16	Accoppiabile nella pipe U, BSWAP non è accoppiabile. Svantaggio: modifica i flag
call	**Tutte le CPU**
	Usare call di tipo near.
call dest1 **jmp dest2**	**286+**
push offset dest2 jmp dest1	Quando CALL è seguita da una jmp, si può cambiare l'indirizzo restituito per la destinazione di jmp.
call dest1 **ret**	**Tutte le CPU**
jmp dest1	Quando CALL è seguita da una RET, la CALL può essere sostituita da un'istruzione jmp.
cbw	**386+**
mov ah, 0	Quando si sa che AL < 128 per ottenere una maggiore velocità si può usare MOV. CBW è però più piccola (1 byte).
cdq	**486+**
xor edx, edx	Se si sa che EAX è positivo. Maggiore velocità e accoppiabilità. Svantaggio: modifica i flag
cdq	**Pentium**
mov edx, eax sar edx, 31	Se il valore di EAX potrebbe essere negativo; maggiore accoppiabilità.
cmp mem, reg	**286**
cmp reg, mem	reg, mem è più veloce di 1 ciclo di CPU.
cmp reg, mem	**386**
cmp mem, reg	mem, reg è più veloce di 1 ciclo di CPU.
dec reg16	**modalità a 16 bit della CPU**
lea reg16, [regl6 - 1]	Conserva i flag BX, BP DI, SI
dec reg32	**modalità a 32 bit della CPU**
lea reg32, [reg32 - 1]	Conserva i flag EAX, EBX, ECX, EDX EDI, ESI, EBP

div <op>	**8088**
shr accum, 1	Quando <op> è una potenza di 2, dividere con uno scorrimento (usare CL for 4, 8 e così via).

div <op>	**186+**
shr accum, n	Quando <op> è una potenza di 2; dividere con uno scorrimento.

enter imm16, 0	**286+**

push bp
mov bp, sp
sub sp, imm16
	ENTER è sempre più lenta e lunga 4 byte.
	Se imm16 = 0, push/mov è più compatta

enter imm16, 0	**386+**

push ebp
mov ebp, esp
sub esp, imm16	

inc reg16	**Modalità a 16 bit della CPU**
lea reg16, [reg16 + 1]	Conserva i flag BX, BP DI, SI

inc reg32	**Modalità a 32 bit della CPU**
lea reg32, [reg32 + 1]	Conserva i flag EAX, EBX, ECX, EDX, EDI, ESI, EBP

int nn	**Modalità reale**
pushf call dword ptr mem	int richiede molti più cicli, 16-82 cicli a seconda della modalità della CPU. CALL DWORD PTR ne richiede solo 4. Prima si deve però salvare una copia del vettore, sapendo che non verrà modificato. Può provocare problemi con i gestori di memoria che devono controllare tutti gli interrupt.

icxz <dest>	**486+**
test cx, cx je <dest>	JCXZ è più veloce e compatta dall'8088 all'80286. Sul 386 ha più o meno la stessa velocità.
test ecx, ecx je <dest>	Non usare mai JCXZ o JECXZ su un 486 o un Pentium se non per ricercare maggiore compattezza.

lea reg, mem	**8088-286**
mov reg, OFFSET mem	MOV reg, imm è più veloce su 8088-286. Dal 386 in poi hanno prestazioni identiche.
Nota:	l'istruzione LEA ha molti altri usi; vedere: add, inc, dec, mov, mul.

leave	**486+**

mov sp, bp
pop bp
 	
mov esp, ebp
pop ebp
	LEAVE è lunga 1 solo byte long ed è più veloce su 186-386.
	MOV-POP è più veloce su 486 e Pentium.

lodsb	**486+**

mov al, [si]
inc si
lodsw
mov ax, [esi]
add si, 2
lodsd
mov eax, [esi]
add esi, 4
	LODS è lunga solo 1 byte in più ed è più veloce su 8088-386, molto più lenta su 486. Sul Pentium LODS richiede 2 cicli di CPU. MOV/INC o MOV/ADD possono essere accoppiate per ottenere 1 ciclo.
	Nota: per incrementare senza utilizzare i flag si può usare LEA SI, [SI+n].
	Nota: se DF=1, usare DEC o SUB.

loop <dest>	**386+**
dec cx jnz <dest> **loop <dest>** dec ecx jnz <dest>	LOOP è più veloce e compatta su 8088-286. Dal 386 in poi DEC/JNZ è molto più veloce.

loopXX <dest> (Xx = e, ne, z o nz)	**486+**
je $+5 dec cx jnz <dest> **loopXX <dest>** je $+5 dec ecx jnz <dest>	Le istruzioni di ciclo condizionale sono molto più lente dal 486 in poi. Sono più compatte e veloci su 8088-286. Su 386 la velocità è approssimativamente la stessa.

mov reg2, reg1	**286+**
seguito da una delle seguenti istruzioni: inc/dec/add/sub reg2 lea reg2, [regl+n]	Più veloce, più compatta e conserva i flag. Si tratta di un metodo per eseguire una MOV e una ADD/SUB di una costante, *n*.

mov acc, reg	**Tutte le CPU**
xchg acc, reg	xchg produce codice più compatto quando si può ignorare un registro.

mov mem, imm	**Pentium**

lea bx, mem
mov [bx], imm
 	
mov ax, imm
mov mem, ax
	I parametri scostamento/immediato pregiudicano l'accoppiabilità. LEA/MOV possono dunque essere utilizzate se fra di esse vengono inserite altre istruzioni per evitare blocchi AGI. MOV/MOV possono essere più facilmente accoppiate.

mov [bx+2], imm	**Pentium**
mov ax, imm mov [bx+2], ax	Maggiore accoppiabilità
lea bx, [bx+2] mov [bx], imm	Maggiore accoppiabilità

movsb	**486+**

mov al, [si]
inc si
 	
mov [di], al
inc di
	MOV è più veloce e compatta per spostare un singolo byte, una word o una dword su 8088-386. Dal 486 in avanti il metodo MOV/INC è più veloce.
	Per spostare un blocco, REP MOVS è sempre più veloce.
	
	Nota: se DF=1, usare DEC o SUB.

movsw	

mov ax, [si]
add si, 2
mov [di], ax
add di, 2	

movsd	

mov eax, [esi]
add esi, 4
mov [edi], eax
add edi, 4	

movzx r16, rm8	**486+**
xor bx, bx mov bl, al	MOVZX è più veloce e compatta sul 386. Dal 486 in avanti XOR/MOV è più veloce. Accoppiabilità sul Pentium (l'origine può essere reg o mem.) Svantaggio: modifica i flag

movzx r32, rm8	
xor ebx, ebx mov bl, al	

movzx r32, rm16	
xor ebx, ebx mov bx, ax	

mul n	**8088+**
shl ax, cl	Usare scorrimenti o ADD invece delle moltiplicazioni se n è una potenza di 2.

mul n	**Pentium**
add ax, ax	ADD si comporta meglio dello scorrimento singolo poiché può essere accoppiata più facilmente.

mul	modalità a 32 bit della CPU
lea lea eax, [eax+eax*4]	Usare LEA per moltiplicare per 2,3,4,5,7,8,9 (per moltiplicare EAX * 5) Sul Pentium, LEA si comporta meglio rispetto a SHL poiché può essere accoppiata in entrambe le pipe; SHL può essere accoppiata solo nella pipe U.
or reg, reg	**Pentium**
test reg, reg	Maggiore accoppiabilità poiché OR scrive sui registri (valido se orig=dest)
pop mem	**486+**
pop reg mov mem, reg	Più veloce dal 486 in poi. Maggiore accoppiabilità sul Pentium.
push mem	**486+**
mov reg, mem push reg	Più veloce su 486. Maggiore accoppiabilità sul Pentium
pushf	**486+**
rcr reg, 1 oppure rcl reg, 1	Per risparmiare solo il flag carry si può usare una rotazione (RCR o RCL) in un registro. RCR e RCL sono accoppiabili (solo nella pipe U) e richiedono 1 solo ciclo di CPU. PUSHF è lenta e non è accoppiabile.
popf	**486+**
rcl reg, 1 oppure rcr reg, 1	Per ripristinare solo il flag carry.
rep movsb	**8088+**
nessuna	Sempre la più veloce.
rep movsw rep movsd rep stosb	**8088+**
nessuna	Sempre la più veloce.

rep stosw **rep stosd** **rep scasb**	**Pentium**

loop1:
mov al, [di]
inc di
cmp al, reg2
je exit
dec cx
jnz loop1
exit:
	REP SCAS è più veloce e compatta su 8088-86. Il codice espanso è più veloce sul Pentium grazie alla possibilità di accoppiare le istruzioni.
	
	Nota: vedere il Capitolo 14.

shl reg, 1	**Pentium**
add reg, reg	ADD è meglio accoppiabile rispetto a SHL che può essere accoppiato solo nella pipe U.

stosb	**486+**
mov [di], al inc di	STOS è più veloce e compatta su 8088-286 e ha la stessa velocità sul 386. Dal 486 in poi MOV/ INC è leggermente più veloce.

stosw
mov [di], ax
add di, 2
stosd
mov [edi], eax
add edi, 4
	REP STOS è più veloce su 8088-386. MOV/INC o MOV/ADD sono più veloci dal 486 in poi.

Nota: usare LEA si, [si+n] per far avanzare LEA senza modificare i flag.

xchg	**Tutte le CPU**
	Usaree xchg acc, reg per eseguire una MOV di 1 byte quando un registro può essere ignorato.

xchg reg1, reg2	**Pentium**
push reg1 push reg2 pop reg1 pop reg2	Per motivi di accoppiabilità, le push e pop sono più veloci sul Pentium. Svantaggio: usa lo stack.

xchg regl, reg2	**Pentium**

mov reg3, regl
mov regl, reg2
mov reg2, reg3
	Più veloce e maggiore accoppiabilità se è disponibile reg3.

xlatb	486+

mov bh, 0
mov bl, al
mov al, [bx]
	XLAT è più veloce e compatta su 8088-386. Le MOV sono più veloci dal 486 in avanti. È sempre meglio ridisporre le istruzioni per evitare blocchi AGI e ottenere l'accoppiabilità sul Pentium.
	Gli azzeramenti devono essere posti prima del ciclo.
	Svantaggio: modifica i flag.

xor ebx, ebx
mov bl, al
mov al, [ebx]	

Appendice C

Principi di ottimizzazione elencati per CPU

Questa appendice contiene varie indicazioni e informazioni di ottimizzazione elencate in ordine di CPU.

C.1 8088

La massima velocità di esecuzione è di un byte di codice ogni quattro cicli di CPU, a causa dell'operazione di prefetch. In realtà la velocità è inferiore a causa delle operazioni di lettura e scrittura sulla memoria, di refresh della memoria DRAM e così via.

La reale velocità di esecuzione è costituita dal massimo tempo di prefetch (4 cicli di CPU per byte) o dalle indicazioni "ufficiali" presentate nella documentazione.

Evitando accessi inutili alla memoria, l'operazione di prefetch può essere eseguita in modo più veloce (poiché si usano i registri).

Ogni accesso al bus richiede quattro cicli: prefetch, lettura o scrittura dei dati (gli accessi a word richiedono 8 cicli).

Gli accessi a 16 bit sono più rapidi rispetto a due operazioni su 8 bit poiché per i secondi 8 bit vi è solo una penalizzazione di 4 cicli. Due operazioni a 8 bit richiedono un ulteriore tempo di prefetch e di esecuzione e aumentano le dimensioni del codice.

È sempre consigliabile conservare i valori nei registri (evitando accessi alla memoria).

È consigliabile utilizzare le istruzioni che hanno una forma più compatta in quanto il collo di bottiglia è spesso rappresentato dall'operazione di fetch.

Si devono evitare i salti in quanto bloccano la coda di prefetch.

L'operazione di prefetch della memoria DRAM rallenta tutte le stime dei tempi di circa il 5-7%.

Se in un ciclo si deve utilizzare un'istruzione che richiede un lungo tempo di esecuzione (ad esempio mul o DIV), è consigliabile posizionarla appena dopo un salto eseguito frequentemente in modo da consentire di riempire la coda di prefetch.

Si devono ridurre le istruzioni rapide dopo un salto. Le istruzioni rapide (da 2 a 4 cicli di CPU) non danno alla coda di prefetch il tempo di riempirsi nuovamente. Se in un ciclo vi è un'istruzione che richiede molti cicli di CPU, si cerchi di posizionarla verso l'inizio del ciclo di istruzioni.

Si deve evitare il calcolo di lunghi indirizzi effettivi; in particolare tali calcoli devono essere posizionati fuori dai cicli in quanto un calcolo di un indirizzo effettivo richiede dai 5 ai 12 cicli.

Quando si accede allo stack utilizzando BP, si deve cercare di utilizzare l'area compresa fra -128 e +127 byte rispetto a BP poiché tali accessi richiedono una minore quantità di codice rispetto a quelli a distanza superiore.

C.2 **286**

Allineare i dati ai quali si deve accedere a 16 bit per volta lungo gli indirizzi pari. L'utilizzo di word ad indirizzi dispari provoca due accessi al bus e almeno due cicli di CPU di penalizzazione.

Ogni accesso a 16 bit al bus richiede due cicli di CPU, più uno per ogni stato di attesa.

Le etichette che fungono da destinazione dei cicli di istruzioni devono essere allineate agli indirizzi pari. Infatti per leggere la prossima istruzione nella coda di prefetch sarà richiesto un minor numero di cicli di fetch. A seconda dell'istruzione che si trova dopo la l'etichetta, il posizionamento a indirizzi dispari può richiedere due o più cicli aggiuntivi.

A causa delle caratteristiche progettuali della scheda di memoria e della velocità della RAM video, il PC/AT IBM può diventare lento quanto un 8088. Le schede video per il PC/AT IBM operano su un bus da 8 Mhz e generalmente aggiungono molti stati di attesa.

I calcoli di indirizzi effettivi richiedono 0 o 1 ciclo. Se l'indirizzo effettivo contiene l'indirizzo base, l'indice e lo scostamento, richiede un ciclo di CPU. Per eliminare questo ciclo di CPU si può eseguire l'unione dei registri all'esterno del ciclo di istruzioni.

Utilizzare le nuove forme di scorrimenti e rotazioni che consentono di eseguire operazioni multiple con una singola istruzione.

C.3 **386**

Per il calcolo degli indirizzi effettivi vale quanto detto per il 286.

Sul 386 le architetture della memoria sono molto varie. Molti sistemi sono dotati di memoria cache e la memoria principale inserisce da 1 a 5 stati di attesa. La memoria

video può prevedere molti stati di attesa. Alcune schede richiedono fino a 32 stati di attesa.

Il codice e i dati dovrebbero essere allineati sui limiti di una dword.

Il 386SX accede ai dati a 32 bit in due operazioni.

Nuovo tipo di indirizzamento a 32 bit per segmenti estesi in modalità protetta:

Base + indice scalato + scostamento;
Base = EAX, EBX, ECX, EDX, EDI, ESI, ESP
Indice=uno qualsiasi dei registri precedenti (escluso ESP quando il valore non viene scalato).
Lo scostamento può essere di 1 o 4 byte.

L'operazione di indirizzamento a 32 bit in un segmento a 16 bit implica un ritardo di un ciclo di CPU per il prefisso.

Utilizzando l'indice scalato all'interno di un ciclo si ottiene codice più lento. Ad esempio:

```
loop1;
        add     eax, array[edi+4]       ;somma la dword da un array
        inc     edi                     ;fa avanzare l'indice all'elemento successivo dell'array
        dec     edx                     ; decrementa il contatore del ciclo
        jnz     loop1
```

Tale codice può essere ottimizzato nel seguente modo:

```
        shl     edi,2                   ;premoltiplicazione di un fattore 4
loop1;
        add     eax, array[edi]
        add     edi, 4
        dec     edx
        jnz     loop1
        shr     edi, 2                  ; elimina, se necessario, il fattore di scala
```

A causa del funzionamento degli operandi a 32 bit, vi sono alcune situazioni interessanti. Le operazioni MOV immediate a 32 bit richiedono sempre 4 byte. Altre istruzioni, ad esempio l'istruzione OR esegue l'estensione del segno:

```
1a.     sub     eax, eax        ; eax = 0       2 cicli, 2 byte
        inc     eax             ; eax = 1       2 cicli, 1 byte
1b.     mov     eax, 1          ; eax = 1       2 cicli, 5 byte
2a.     or      eax, -1         ; eax = -1      2 cicli, 3 byte
2b.     mov     eax, -1         ; eax = -1      2 cicli, 5 byte
```

C.4 486

Grazie alla presenza della memoria cache interna, è possibile eseguire molte più ottimizzazioni rispetto ai sistemi precedenti.

Le procedure richiamate frequentemente e le destinazioni dei salti devono essere allineate ai limiti di gruppi di 16 byte poiché queste sono le dimensioni di una linea di cache (come minimo si devono allineare sui limiti di 4 byte).

I dati devono essere allineati sulla base delle loro dimensioni, ovvero le word devono essere allineate alle word, le dword alle dword e così via.

Il 486SX ha lo stesso bus a 32 bit e gli stessi criteri di allineamento del 486 ma non contiene l'unità FPU.

Calcolo dell'indirizzo effettivo

Utilizzando due registri (base più indice)	ritardo di un ciclo.
Utilizzando uno scostamento	ritardo di un ciclo.
Utilizzando un indice scalato	ritardo di un ciclo.

È preferibile utilizzare istruzioni semplici (definite come tali per il Pentium). Tali istruzioni richiedono in genere un solo ciclo di CPU sul 486. Le istruzioni complesse, come LOOP, XLAT e XCHG sono molto più lente.

C.5 Blocchi AGI (Address Generation Interlock)

Due cicli di ritardo in modalità reale (un ciclo se si trova a due istruzioni di distanza)
Un ciclo di ritardo in modalità protetta

I blocchi AGI su SP ed ESP vengono risolti internamente e dunque non vi sono ritardi per le istruzioni PUSH, POP, CALL e RET.

Utilizzando SP o ESP come un registro generale, si provocherà comunque un blocco AGI:

```
add     esp, 10
```

La nuova istruzione BSWAP è veloce e non modifica il contenuto dei flag. In particolare occorre notare che le istruzioni ROR e BSWAP non eseguono la stessa operazione anche se entrambe consentono di accedere alla word alta di EAX tramite AX oppure AL e AH.

```
ror     eax, 16         ; 2 cicli AABBCCDD —> CCDDAABB
bswap                   ; 1 ciclo AABBCCDD —> DDCCBBAA
```

Gli scorrimenti e le rotazioni immediate con valori da 2 a 31 sono più veloci rispetto alle stesse operazioni eseguite con il valore 1. Il terzo byte di uno scorrimento immediato può accettare il valore 1 ma l'assembler non lo genera. Questo è valido solo per il 486.

```
shl     eax, 1          ; 3 cicli, 2 byte
shl     eax, 2          ; 2 cicli, 3 byte
```

Si applica alle istruzioni: SHL, SHR, SAL, SAR, ROL e ROR

C.6 **Pentium**

Le procedure richiamate frequentemente e le destinazioni dei salti devono essere allineate ai limiti di 32 byte poiché queste sono le dimensioni di una linea di cache. Questo accorgimento non è importante come sul 486 grazie alla previsione della destinazione dei salti. Ma l'importanza può essere notevole poiché una procedura compatta e frequentemente richiamata può, se allineata correttamente, rientrare completamente in una linea di cache mentre in caso di disallineamento richiede due linee di cache. D'altra parte, l'allineamento di ogni procedura e ciclo di istruzioni provoca uno spreco di byte contenenti istruzioni NOP.

I dati devono essere allineati secondo le loro dimensioni: word allineate a word e dword allineate a dword.

Il calcolo di un indirizzo effettivo richiede sempre 0 cicli.

Si deve cercare di utilizzare istruzioni semplici per far lavorare entrambe le pipeline.

Seguire le regole presentate nelle Appendici D ed E.

Utilizzare le forme a uno o due cicli di CPU delle istruzioni accoppiabili; le forme a 3 cicli bloccano i punti di esecuzione. Le forme a due cicli bloccano un punto di esecuzione solo se non vengono accoppiate a un'altra istruzione da due cicli.

Non si verifica alcun accoppiamento di istruzioni quando tali istruzioni contengono uno scostamento e un valore immediato come in:

```
mov    mem1, 5          ; 1 ciclo, nessuna parità

add    array[bx], 1     ; 1 ciclo, nessuna parità
```

Il seguente esempio mostra come è possibile ottimizzare alcune situazioni scostamento/immediato per consentirne l'accoppiamento con un'istruzione vicina:

```
mov    mem1, 0          ; queste due istruzioni non sono accoppiabili
mov    mem2, 0          ; e richiedono 1 ciclo ciascuna
```

Sostituire con:

```
xor    eax, eax         ; 1 ciclo (accoppiabile all'istruzione successiva)
(pairable, instruction) ; 0 cicli posizione utilizzabile
mov    mem1, eax        ; 1 ciclo (accoppiabile alla successiva)
mov    mem2, eax        ; 0 cicli
```

Si devono ottimizzare (accoppiando le istruzioni) solo i cicli eseguiti nella memoria cache. Se un ciclo non verrà eseguito molte volte, è probabilmente consigliabile ottimizzarne le dimensioni per far rientrare altro codice nella cache.

Il sistema di previsione della destinazione dei salti consente di ridurre i salti che si comportano in modo regolare a un solo ciclo. La penalizzazione in caso di previsione errata è pesante:

Istruzione	Penalizzazione nella pipe U	Penalizzazione nella pipe V
Jcc	4 cicli	5 cicli
call near	3 cicli	3 cicli
jmp near	3 cicli	3 cicli

Appendice D

Istruzioni semplici accoppiabili sul Pentium

Formato istruzioni	Esempio a 16 bit	Esempio a 32 bit
MOV reg, reg	mov ax, bx	mov eax, edx
MOV reg, mem	mov ax, [bx]	mov eax, [edx]
MOV reg, imm	mov ax, 1	mov eax, 1
MOV mem, reg	mov [bx], ax	mov [edx], eax
MOV mem, imm	mov [bx], 1	mov [edx], 1
alu reg, reg	add ax, bx	cmp eax, edx
alu reg, mem	add ax, [bx]	cmp eax, [edx]
alu reg, imm	add ax, 1	cmp eax, 1
alu mem, reg	add [bx], ax	cmp [edx], eax
alu mem, imm	add [bx], 1	cmp [edx], 1
dove alu = add, adc, and, or, xor, sub, sbb, cmp, test		
INC reg	inc ax	inc eax
INC mem	inc var1	inc [eax]
DEC reg	dec bx	dec ebx
DEC mem	dec [bx]	dec var2
PUSH reg	push ax	push eax
POP reg	pop ax	pop eax
LEA reg, mem	lea ax, [si+2]	lea eax, [eax+4*esi+8]
JMP near	jmp etichetta	jmp etichetta2
CALL near	call proc	call proc2
Jcc near	jz etc	jnz etc2
NOP	nop	nop
shift reg, 1	shl ax, 1	rcl eax, 1
shift mem, 1	shr [bx], 1	rcr [ebx], 1
shift reg, imm	sal ax, 2	rol esi, 2
shift mem, imm	sar ax, 15	ror [esi], 31

- Le istruzioni rcl e rcr non sono accoppiabili se il conteggio è diverso da 1.
- Tutte le istruzioni memoria-immediato (mem, imm) non sono accoppiabili con uno scostamento nell'operando in memoria.
- Le istruzioni con registri di segmento non sono accoppiabili.

Appendice E

Accoppiamento di istruzioni: regole per il Pentium

1. Entrambe le istruzioni devono essere semplici (vedere l'Appendice D).
2. Gli scorrimenti e le rotazioni (SHL, SHR, SAL, SAR, ROL, ROR, RCL o RCR) sono accoppiabili sono nella pipe U.
3. ADC e SBB sono accoppiabili solo nella pipe U.
4. JMP, CALL e Jcc sono accoppiabili solo nella pipe V (Jcc = Jump on condition code).
5. Nessuna delle istruzioni può contenere uno scostamento e un operando immediato. Ad esempio:

```
mov [bx+2], 3   ; 2 è uno scostamento, 3 è una costante (valore immediato)
mov mem1, 4     ; mem1 è uno scostamento, 4 è una costante (valore immediato)
```

6. Le istruzioni con prefisso sono accoppiabili solo nella pipe U. Fra queste vi sono anche le istruzioni che iniziano con 0Fh tranne il caso speciale dei salti condizionali a 16 bit ma solo a partire dal 386. Esempi di istruzioni con prefisso:

```
mov ES:[bx], 1
mov eax, [si]   ; operando a 32 bit in un segmento di codice a 16 bit
mov ax, [esi]   ; operando a 16 bit in un segmento di codice a 32 bit
```

7. L'istruzione nella pipe U deve essere lunga 1 byte o non potrà essere accoppiata se non a partire dalla seconda esecuzione, in cui verrà eseguita dalla memoria cache.
8. Fra le istruzioni non sono consentite dipendenze di registri di tipo scrivi-leggi o scrivi-scrivi tranne per i casi speciali del registro dei flag e del puntatore allo stack (regole 9 e 10).

```
mov ebx, 2      ; scrive su EBX
add ecx, ebx    ; legge EBX ed ECX, scrive su ECX
                ; EBX viene letto dopo essere stato scritto, istruzioni non accoppiabili

mov ebx, 1      ; scrive su EBX
mov ebx, 2      ; scrive su EBX
                ; EBX viene scritto dopo essere stato scritto, istruzioni non accoppiabili
```

9. L'eccezione del registro dei flag consente di accoppiare un'istruzione ALU con un'istruzione Jcc anche se l'istruzione ALU modifica i flag e Jcc legge i flag. Ad esempio:

```
cmp al, 0      ; CNP modifica i flag
je addr        ; JE legge i flag, ma è accoppiabile

dec cx         ; DEC modifica i flag
jnz loop1      ;JNZ legge i flag, ma è accoppiabile
```

10. L'eccezione del puntatore allo stack consente di accoppiare due PUSH o due POP anche se entrambe leggono o scrivono il registro SP (o ESP).

```
push eax       ; ESP viene letto e modificato
push ebx       ; ESP viene letto e modificato, ma rimane accoppiabile
```

Appendice F

Istruzioni composte da un unico byte

Istruzioni di 1 byte - accoppiabili

dec	reg	DECrement register (16-bit o 32-bit)
inc	reg	INCrement register (16-bit o 32-bit)
nop		NO Operation
pop	reg	POP registro (registro generale a 16 o 32 bit)
push	reg	PUSH registro (registro generale a 16 o 32 bit)

Istruzioni di 1 byte - non accoppiabili

aaa	Ascii Adjust after Addition
aas	Ascii Adjust after Subtraction
cbw	Convert Byte to Word
cdq	Convert Double to Quad
clc	CLear Carry flag
cld	CLear Direction flag
cli	CLear Interrupt flag (disabilita gli interrupt mascherabili)
cmc	CoMplement Carry flag
cmpsb	CoMPare String Byte
cmpsw	CoMPare String Word
cmpsd	CoMPare String Dword
cwd	Convert Word to Double
cwde	Convert Word to Double Extended
daa	Decimal Adjust after Addition
das	Decimal Adjust after Subtraction
hlt	HaLT
in acc, dx	INput from port
insb	INput from port to String Byte
insw	INput from port to String Word
insd	INput from port to String Dword

int 3	software INTerrupt 3
into	INTerrupt on Overflow
iret	Interrupt RETurn
iretd	Interrupt RETurn Double
lahf	Load Flags into AH register
leave	LEAVE high level procedure
lodsb	LOaD String Byte
lodsw	LOaD String Word
lodsd	LOaD String Dword
movsb	MOVe String Byte
movsw	MOVe String Word
movsd	MOVe String Dword
out dx, acc	OUTput to port
outsb	OUTput String to port Byte
outsw	OUTput String to port Word
outsd	OUTput String to port Dword
pop sreg	POP segment register (solo DS, ES, SS)
popa	POP All
popad	POP All Double
popf	POP Flags
popfd	POP Flags Double
push sreg	PUSH segment register (solo DS, ES, SS)
pusha	PUSH All
pushad	PUSH All Double
pushf	PUSH Flags
pushfd	PUSH Flags Double
ret	RETurn from procedure
retn	RETurn from procedure Near
retf	RETurn from procedure Far
sahf	Store AH into Flags
scasb	SCAn String Byte
scaws	SCAn String Word
scasd	SCAn String Dword
stc	SeT Carry flag
std	SeT Direction flag
sti	SeT Interrupt flag (lascia attivi gli interrupt mascherabili)
stosb	STOre String Byte
stosw	STOre String Word
stosd	STOre String Dword
wait	WAIT for co-processor
xchg acc, reg	eXCHanGe accumulatore con il registro (16 o 32 bit)
xlat	TRANSLATe

Byte di prefisso

26h	ES:	uscita dal segmento ES
2Eh	CS:	uscita dal segmento CS
36h	SS:	uscita dal segmento SS
3Eh	DS:	uscita dal segmento DS
64h	FS:	uscita dal segmento FS
65h	GS:	uscita dal segmento GS
66h		prefisso dimensionale dell'operando
67h		prefisso dimensionale dell'indirizzo
FOh	lock	blocca il bus Il blocco funziona solo con le seguenti istruzioni quando queste accedono a un operando in memoria: bts, btr, btc, add, or, adc, sbb, and, sub, xor not, neg, inc, dec, cmpxchg, xadd, xchg (il blocco è automatico)
F2h	repne	REPeat string while Not Equal (anche repnz)
F3h	repe	REPeat string while Equal (anche repz)
	rep	REPeat string (come repe)
OFh		prefisso per codici operativi composti da 2 byte per le nuove istruzioni, come: SETcc, LFS, LGS, CMPXCHG, XADD, Jcc (16 bit), SHLD, BSWAP
D8h-DFh		Istruzioni in virgola mobile

Appendice G

Guida di riferimento rapido dei tempi di esecuzione

MOV

	8088	186	286	386	486	Pentium
mov reg, reg	2	2	2	2	1	1 UV
mov mem, reg	19	19	9	2	1	1 UV
mov reg, mem	18	18	12	4	1	1 UV
mov mem, imm	20	20	13	2	1	1 UV*
mov reg, imm	4	4	4	2	1	1 UV

* = non accoppiabili se vi è uno scostamento e un immediato

POP PUSH

	8088	186	286	386	486	Pentium
pop reg	12	10	5	4	1	1 UV
pop sreg	12	10	5/20	7/21	3/9	3/12 NP
	(cicli di CPU per modalità reale/protetta)					
push reg	15	10	3	2	1	1 UV
push sreg	14	10	3	2	3	1 NP

MFSC

	8088	186	286	386	486	Pentium
xchg reg,reg	4	4	3	3	3	3 NP
xchg reg,mem	31	17	5	5	5	3 NP
clc, stc	2	2	2	2	2	2 NP
cbw	2	2	2	3	3	3 NP
nop	3	3	3	3	3	1 UV
xlat	11	11	5	5	4	4 NP
lea	7	6	3	2	1/2*	1 UV

* = 486: quando EA contiene un registro indice, LEA richiede due cicli

ADD ADCCC SUB SBB AND XOR OR (alu)

	8088	186	286	386	486	Pentium
alu reg, reg	3	3	2	2	1	1 UV
alu mem, reg	30	10	7	7	3	3 UV
alu reg, mem	18	10	7	6	2	2 UV
alu mem, inn	20	16	7	7	3	3 UV*
alu reg, inn	4	4	3	2	1	1 UV

* = non accoppiabile se vi è uno scostamento e un immediato.

SHL SHR SAL SAR RCL RCR ROL ROR (sh)

	8088	186	286	386	486	Pentium
sh reg, 1	2	2	2	3	3	1 PU
sh reg, c1	8/4	5/1	5/1	3	3	4 NP
sh reg, imm	-	5/1	5/1	3	2	1 PU*
sh mem,1	29	15	7	7	4	3 PU
sh mem, c1	32/4	8/1	8/1	7	4	4 NP
sh mem, imm	-	8/1	8/1	7	4	3 PU*

cicli = base/per ogni scrrimento
* = non accoppiabile se vi è uno scostamento e un immediato
rcl, rcr non accoppiabili con conteggi immediati diversi da 1.

JUMP LOOP CALL RET

	8088	186	286	386	486	Pentium
jmp short/near	15	13	8	8	3	1 PV
jmp reg	11	11	8	8	5	2 NP
jmp mem	23	17	12	11	5	2 NP
Jcc	4/16	4/13	3/8	3/8	1/3	1 PV
jcxz	6/18	5/16	4/9	5/10	5/8	5/6 NP
loop	5/17	5/15	4/9	11	6/7	5 NP
loope	6/18	6/16	4/9	11	6/9	7 NP

(cicli di CPU per nessun salto/salto)

call	near	23	14	8	8	3 1 PV
call	reg	20	13	8	8	5 2 NP
call	mem	35	19	12	11	5 2 NP
ret	near	20	16	12	11	5 2 NP

Stringhe

	8088	186	286	386	486	Pentium
rep movs	17	8	4	4	3	1 NP
rep stos	10	9	3	5	4	1 NP
repecmps	22	22	9	9	7	4 NP
repescas	15	15	8	8	5	4 NP
lodsb	16	10	5	5	5	2 NP
stosb	15	10	3	4	5	3 NP

Combinazioni importanti per il Pentium

	8088	186	286	386	486	Pentium
dec reg/Jcc	7/19	7/16	5/10	5/10	2/4	1 accoppiabile
	(cicli per: nessun salto/salto)					
mov r, m/inc reg	21	21	14	6	2	1 accoppiabile
mov m, r/inc reg	21	21	11	4	2	1 accoppiabile
push reg/pop reg	27	20	8	6	2	1 accoppiabile
xchg reg,reg	4	4	3	3	3	3 NP

Appendice H

Registri non documentati per il Pentium

La natura sofisticata dell'architettura del Pentium ha spinto Intel a sviluppare una serie di registri interni e contatori utilizzabili per vari scopi, ad esempio la ricerca di errori e la verifica delle prestazioni. Per motivi ignoti, Intel ha deciso di non pubblicare queste informazioni.

Questi registri "segreti" possono essere letti e scritti rispettivamente con le istruzioni RDMSR e WRMSR. Ecco cosa dice la documentazione Intel a proposito dell'istruzione RDMSR nel *Pentium Processor User's Manual, Volume 3: Architecture and Prorgramming Manual*:

"RDMSR consente di leggere il contenuto di registri specifici del modello che controllano le funzioni di test, trace dell'esecuzione, controllo delle prestazioni e verifica degli errori della macchina. Per ulteriori informazioni consultare il Pentium Processor Data Book. I valori 3h, 0Fh e i valori oltre il 13h sono riservati. Non eseguire RDMSR inserendo in ECX un valore riservato".

RDMSR ReaD Machine Specific Register
Input: registro numerico ECX

ECX	Nome	Descrizione
0	Indirizzo di verifica della macchina	Indirizzo del ciclo che ha provocato l'eccezione
1	Tipo di verifica della macchina	Tipo del ciclo che ha provocato l'eccezione

Per altri valori utilizzati per controllare le prestazioni della memoria fascia, per il test di TLB e BTB e per il controllo delle prestazioni, consultare l'Appendice H (nota: qui si intende l'Appendice H del Intel Pentium Processor User's Manual).

L'Appendice H del manuale Intel è composta da poche righe di testo e informa solo del fatto che "vi sono informazioni non essenziali riguardanti il processore Pentium che devono essere considerate confidenziali e che pertanto non vengono documentate in questa pubblicazione ".

Certamente si tratta di informazioni confidenziali, almeno fino a quando si scopre che tutti i concorrenti le utilizzano. L'autore non ha mai visto e non conosce il contenuto dell'Appendice H e per ottenere una copia occorre contattare Intel e firmare un'apposita licenza.

Grazie al lavoro di Terje Mathisen, uno sviluppatore di software norvegese, alcune delle informazioni necessarie per l'ottimizzazione delle prestazioni sono state recentemente rese pubbliche.

È possibile accedere contemporaneamente al valore dei due contatori MSR 11 h controlla quale dei due contatori è disponibile. I due contatori possono essere letti come MSR 12 h e MSR 13 h. Vengono utilizzati solo i 32 bit inferiori di MSR 11 h. I primi 16 bit controllano MSR 12 h e i 16 di successivi controllano MSR 13 h. La codifica di questi due gruppi di 16 bit è identica:

Bit	Descrizione
0-5	numero del contatore (vedere Tabella H.1)
6	1=contatore negli anelli 0, 1, 2
7	1=contatore nell'anello 3
8	0=contatore eventi, 1=contatore cicli
9-15	riservato, non modificare

Ad esempio, per contare il numero di istruzioni eseguite in ciascuna pipe si devono utilizzare i contatori 16 h e 17 h nel seguente modo:

Listato

Le istruzioni RDMSR e WRMSR sono privilegiate. Devono essere eseguite dall'anello 0 in modalità protetta o reale. È possibile che vi sia un modo per consentire l'esecuzione di queste istituzioni anche negli anelli 1, 2 o 3 della modalità protetta o in modalità virtuale 8086, ma, naturalmente, si tratta di informazioni riservate e non documentate.

```
cn_1 ecu 16h            ; istruzioni totali eseguite
cn_2 equ 17h            ; istruzioni eseguite nella pipe V
ev3 eou SOh             ; numero eventi nell'anello 3

mov ecx, 11h
rdmsr
and eax, 0FE00FEh       ; salva i bit riservati
or eax, (cn_1+ev3) + ((cn2+ev3) sh1 16)
wrmsr

; salva i valori iniziali dei contatori

movecx, 12h
rdmsr
mov ctr_12, eax
mov ctr_12 [2], edx

mov ecx, 13h
rdmsr
mov ctr_13, eax
mov ctr_13[2], edx

; « inserire il codice di test »

; determina la differenza
```

```
mov ecx, 12h
rdmsr
sub eax, ctr_12
sbb edx, ctr_12[2]
mov ctr_12, eax
mov ctr_12 [2], edx

mov ecx, 13h
rdmsr
sub eax, ctr_13
sbb edx, ctr_13[2]
mov ctr_13, eax
mov ctr_13 [2], edx
```

Tabella H.1 Registri contatori non documentati sul Pentium.

Numero contatore	Nome
0	data read
1	data write
2	data TLB miss
3	data read miss
4	data write miss
5	write (hit) to M or E state lines
6	data cache lines written back
7	data cache snoops
8	data cache snoops hits
9	memory accesses in both pipes
A	bank conflicts (può provocare il blocco della macchina)
B	misaligned data memory references
C	code read
D	code TLB miss
E	code cache miss
F	any segment register load
12	branches
13	BTB hits

(continua)

Tabella H.1 *(continua)*

Numero contatore	Nome
14	branch taken or BTB hit
15	pipelines flushes
16	instructions executed
17	instructions executed in the v-pipe
18	bus utilization (clock)
19	pipelines stalled by write backup
1A	pipeline stalled by data memory read
1B	pipeline stalled by write to E or M line
1C	locked bus cycle
1D	I/O read or write cycle
1E	noncacheable memory references
1F	AGI
22	floating-point operations
23	breakpoint 0 match
24	breakpoint 1 match
25	breakpoint 2 match
26	breakpoint 3 match
27	hardware interrupts
28	data read or write
29	data read miss or data write miss

Appendice I

Riepilogo dei comandi di DEBUG32

A seconda della modalità di CPU impiegata possono essere utilizzati indirizzi nei seguenti formati:

segmento:offset segmento in modalità reale più l'offset;
selettore|offset selettore in modalità protetta più l'offset;
lineare+offset indirizzo lineare in modalità protetta più l'offset.

Gli indirizzi a segmento (identificatore ":") fanno riferimento a un indirizzo di segmento sia in modalità reale che in modalità protetta. Gli indirizzi a selettore (identificatore "|") devono essere utilizzati in modalità protetta per far riferimento a un valore "selettore + offset".

Nei comandi "|" separa le opzioni e le parentesi angolari "<>" indicano un parametro opzionale.

Assemble <indirizzo>
Assembla le istruzioni in memoria.

BP <indirizzo> <R | W | RW | I>
Imposta un breakpoint. Se non vengono specificate le opzioni R/W/RW/I, il breakpoint è solo per l'esecuzione.
R Interruzione nella lettura dei dati.
W Interruzione sulla scrittura dei dati.
RW Interruzione in lettura o scrittura.
I Utilizza i registri di debug per definire un'interruzione durante l'esecuzione (funziona solo in memoria ROM).

BC <indirizzo|*>
Cancella un breakpoint ("*" per tutti i breakpoint).

BL
Elenca i breakpoint. Il suo formato è:

n) e address t=n

"t=n" corrisponde alle esecuzioni totali.

CLS
Cancella lo schermo

CPU
Visualizza il tipo di CPU, la modalità operativa, lo stato A 20 e la lunghezza di prefetch.

Compare <indirizzo1 <indirizzo2> <L lunghezza>>
Confronta i dati in indirizzo1 con i dati in indirizzo2 per la lunghezza specificata in L lunghezza.

DA <indirizzo>
Stampa in formato ASCII i dati che si trovano in indirizzo.

DB <indirizzo>
Stampa in formato byte i dati che si trovano in indirizzo.

DD <indirizzo>
Stampa in formato double-word (32-bit) i dati che si trovano in indirizzo.

DF <indirizzo>
Stampa in formato puntatore far (16:32) i dati che si trovano in indirizzo.

DP <indirizzo>
Stampa in formato puntatore (16:16) i dati che si trovano in indirizzo.

DW <indirizzo>
Stampa in formato word (16-bit) i dati che si trovano in indirizzo.

Dump <indirizzo>
Stampa nel formato corrente i dati che si trovano in indirizzo.

Enter <indirizzo <byte<,byte>>> 'stringa_di_caratteri'
Immette i dati in memoria.

Fill <indirizzo <fine_indirizzo <byte, <byte<,byte<...>>>>> <L lunghezza>
Riempie un'area di memoria con un determinato valore.

Flip ON|OFF
L'opzione standard ON provoca la commutazione nello schermo dell'applicazione con i comandi T e P. Utilizzando OFF si potrebbe ottenere una visualizzazione confusa dello schermo dell'applicazione e del debugger.

Go <=indirizzo> <temp_breakpoint <temp_breakpoint2 <...>>>
Riprende l'esecuzione.

<=indirizzo>	Imposta il nuovo valore di IP
<temp_breackpoint>	Impostano breakpoint temporanei.

HELP <lettere_iniziali del_comando>
I comandi come HELP o ? producono uno schermo di aiuto.
L'output può essere limitato specificando la le prime lettere dei comandi che devono essere descritti.

Hex valore1 valore2
Calcola la somma esadecimale e la differenza fra valore1 e valore2.

In <indirizzo>
Legge in input un byte dalla porta indirizzo e quindi lo visualizza in esadecimale.

Load <nome_programma <indirizzo>>
Carica il programma specificato.
Per file in formato non eseguibile può essere specificato un indirizzo.

Load <indirizzo>
Carica il file identificato da un commando NAME.
Per file in formato non eseguibile può essere specificato un indirizzo.

Load indirizzo numero_drive primo_settore numero_settori
Carica dati in un settore leggendo dal disco specificato.
numero_drive è 0 per A, 1 per B, 2 per C, ecc.
primo_settore è il settore iniziale del disco (in esadecimale)
numero_settori è il numero di settori da 512 byte (in esadecimale).

LOG <nomefile>
Copia tutto l'output del programma sul file specificato.

LOG senza il nome di un file chiude il file.
L'utilizzo di questa funzionalità è limitato ad aree di codice non-DOS e non interrupt.
L'utilizzo all'interno del DOS o di un gestore di interrupt non è consentito.

Memory
Mostra le allocazioni di blocchi di controllo della memoria.

MORE ON|OFF
L'impostazione standard ON interrompe l'output del programma ad ogni pagina dello schermo.
L'impostazione OFF lascia scorrere l'output. Questa opzione è utile quando i dati devono semplicemente essere scritti sul file e non letti.
Quando si immette il comando MORE, premendo un tasto verrà visualizzata una nuova pagina. Con ESC si esce dal comando e con una cifra vengono visualizzate nuove righe di output.

Move <indirizzo1> <indirizzo2> <L_lunghezza>
Copia i dati in indirizzo1 in indirizzo2 per la lunghezza specificata in L lunghezza.

Name <nome programma I parametri>
Imposta il nome del programma per un successivo comando Load
Imposta i parametri della riga di comando per un programma già caricato.
Out <indirizzo> <valore>
Produce in output il valore sulla porta indirizzo.

PRegs
Visualizza i registri privilegiati.

Ptrace <=indirizzo> <cont>
Traccia singole istruzioni in-line o procedure.
<= indirizzo> imposta l'indirizzo di rientro.
<cont> specifica il numero di istruzioni da tracciare.

Quit
Esce dal debugger e ritorna al DOS.

R <nome_registro <nuovo_valore>>
R senza parametri visualizza tutti i registri nella modalità (16/32) corrente.
R <nome_registro> visualizza il valore corrente di registro e richiede l'immissione di un nuovo valore.
R <nome_registro> <nuovo_valore> assegna un nuovo valore al registro.

R16/R32 ...
R16 imposta la modalità "registri a 16 bit".
R32 imposta la modalità "registri a 32 bit".
Entrambi i comandi eseguono lo stesso comando "R" e in più impostano la modalità a 16/32 bit.

RC
Visualizza i registri modificati.

Read <nome_file <indirizzo>>
Legge il file specificato. Si può specificare un indirizzo oppure si può utilizzare l'impostazione standard "area di prefisso più 100h". La formattazione dei file eseguibili non viene considerata e pertanto tali file vengono letti come se fossero comuni dati.

REAson
Visualizza il motivo dell'ingresso nel debugger.

Search <indirizzo <fine_indirizzo <byte<,byte<,...>>>> <L lunghezza>
Ricerca il byte o il carattere specificato.

SELector value
Visualizza l'indirizzo lineare attivo per il settore specificato.

SKIPCR
Disabilita l'accesso ai registri di controllo e di debug per gli ambienti che non consentono tali accessi.

Trace <=indirizzo> <cont>
Traccia una singola istruzione.
<=indirizzo> Modifica l'indirizzo di rientro.
<cont> Definisce il numero di istruzioni da tracciare.

Unassemble <indirizzo>
Interpreta la memoria a <indirizzo> come un'istruzione assembler.

USE32
USE16
Fa in modo che i comandi Assemble/Unassemble assumano il formato specificato.

Viewswap
Passa allo schermo dell'applicazione. Per tornare al debugger basta premere un tasto. Vedere anche il comando FLIP.

Write indirizzo numero_drive primo_settore numero_settori
Scrive i dati su un settore sulla base del disco specificato.
numero_drive è 0 per A, 1 per B, 2 per C, ecc.
primo_settore è il settore iniziale del disco (in esadecimale)
numero_settori è il numero di settori di 512 byte (in esadecimale)

XA num_pagine
Alloca num_pagine di memoria EMS.

XD handle
Dealloca l'handle EMS specificato.

XM pagina_fisica pagina_logica handle
Mappa l'handle EMS e la pagina logica in una pagina fisica.

XS
Visualizza lo stato EMS corrente.

* commento
Consente di immettere un commento.

Appendice J

Miglioramento delle prestazioni

In questa appendice si parlerà di due metodi hardware che consentono di migliorare le prestazioni globali per i futuri progetti di CPU 80x86:

- nuove istruzioni della CPU;
- nuove architetture 80x86.

J.1 Nuove istruzioni

Uno dei metodi che consentono di migliorare le prestazioni potrebbe essere l'aggiunta di nuove istruzioni. È opinione dell'autore che il concetto RISC (ovvero l'idea originale di ridurre il numero di istruzioni e non il nuovo concetto che richiede un'architettura di tipo "carica/memorizza") sia la direzione errata. Poiché oggi i chip contengono un numero di transistor molto superiore rispetto al momento in cui fu concepita l'architettura RISC, sembra che vi sia un blocco mentale contro l'aggiunta di nuove istruzioni. Naturalmente occorre sempre tenere in considerazione i problemi di compatibilità ma quando si sta scrivendo un programma, spesso ci si trova a pensare: " Ma perché non c'è un'istruzione che esegue questa operazione? ".

Parlando di applicazioni del futuro, probabilmente si pensa alla grafica, ai filmati video e al riconoscimento della voce e della scrittura. Anche se l'introduzione di istruzioni specifiche può aumentare notevolmente le prestazioni di queste applicazioni, è probabilmente conveniente utilizzare hardware specializzato, come ad esempio gli acceleratore grafici. Ma in generale tutti i dispositivi portatili e di basso costo dovranno contenere quante più funzionalità possibili all'interno della CPU.

Come esercizio si può provare a realizzare alcune istruzioni che potrebbero essere implementate come istruzioni semplici (ad un ciclo) accoppiabili sul Pentium. Queste istruzioni dovrebbero essere utilizzate nella maggior parte delle applicazioni odierne e dovrebbero fornire un aumento di prestazioni pari almeno al 100% rispetto ai metodi di esecuzione standard.

Ecco alcuni esempi di istruzioni realizzate dall'autore.

La prima istruzione è CMPI (CoMPare Ignore case). Questa istruzione cerca di risolvere le "carenze alfabetiche" di tutte le CPU. Probabilmente ciò è dovuto al timo-

re che emergono standard alternativi rispetto all'ASCII. Il problema è che in questo modo si sono prodotti migliaia di milioni di CPU senza alcun concetto specifico di carattere a parte quello di byte. Al contrario si potrebbe caricare nella CPU una tabella di 256 byte per consentire l'uso di altri set di caratteri. Le istruzioni potrebbero eseguire conversioni fra lettere maiuscole e minuscole, determinare se una lettera è maiuscola o e minuscola ed eseguire altre operazioni specifiche dei caratteri ASCII.

L'istruzione successiva è CMPL (CoMPare List).

Invece di scrivere questo codice:

```
cmp   al, 10
je    lf
cmp   al, 13
je    cr
cmp   al, 32
je    space
cmp   al, 0
je    null
```

si potrebbe scrivere:

```
cmp1  ebx, al
je    match
```

L'istruzione consente di confrontare con AL ogni byte di EBX. Questa istruzione sarebbe quattro volte più veloce del metodo corrente. Altre forme potrebbero essere:

```
cmp1  ebx, ax
cmp1  bx, al
```

L'istruzione CMPL potrebbe essere combinata con l'istruzione CMPI formando l'istruzione CMPLI (CoMPare List Ignore case).

Un'altra istruzione è CMPOR (CoMPare OR). Questa istruzione dovrebbe confrontare ogni byte di un registro fino a trovare un'eventuale corrispondenza. Questo consentirebbe di confrontare contemporaneamente 4 byte (l'istruzione CMP è un AND logico per il confronto di byte). Il seguente codice confronta 4 byte in modo sequenziale:

```
cmp   al, bl
je    found
cmp   ah, bh
je    found
ror   eax, 16
ror   ebx, 16
cmp   al, bl
je    found
cmp   ah, bh
je    found
```

Utilizzando l'istruzione CMPOR si potrebbe scrivere la seguente forma:

```
cmpor eax, ebx
je    found
```

Si potrebbe anche creare un'istruzione che ignora la differenza fra maiuscole e minuscole creando CMPORI.

Questi non sono che alcuni esempi suggeriti dai problemi che si presentano quando si elabora il testo con le varianti dell'istruzione CMP. Un aspetto importante di queste istruzioni è il fatto che consentono di utilizzare la piena potenza dei 32 bit su dati a 8 bit. Infatti, anche se i microprocessori negli ultimi 20 anni sono passati dagli 8 ai 32 bit, l'elaborazione del testo e dei caratteri viene ancora svolta con istruzioni che elaborano solo 8 bit per volta.

J.2 Altre macchine della classe Pentium

Al momento in cui verrà stampato questo manuale, sul mercato potrebbero esservi altre CPU della classe Pentium progettate da altre società, ad esempio l'AMD (Advanced Micro Devices) K5, il Cyrix M1 e il NexGen Nx586. Senza dubbio questa competizione consente di ottenere almeno due risultati: offrire più scelte e abbassare i prezzi. Il problema per i programmatori che desiderano ottimizzare al massimo le prestazioni dei propri programmi è: in che modo queste CPU gestiscono il codice ottimizzato per il Pentium? Probabilmente nessuno lo sa con certezza se non i progettisti di tali chip.

NexGen afferma che il suo Nx586 supera di gran lunga il Pentium nell'esecuzione di istruzioni intere (a parità di clock). Non è difficile pensare al modo in cui si è ottenuto questo risultato. Basta aumentare il numero di istruzioni semplici e pertanto accoppiabili in entrambe le pipe. Naturalmente i progettisti possono ottimizzare molti altri dettagli, come ad esempio le prestazioni della memoria cache, la previsione della destinazione dei salti, le prestazioni dei buffer di I/O e così via. Poiché questi fattori possono variare significativamente, è sempre consigliabile eseguire un test delle porzioni più critiche del codice su tutte le principali piattaforme.

L'architettura interna dell'Nx586 è abbastanza diversa da quella del Pentium. Durante l'operazione di fetch delle istruzioni, ogni istruzione viene convertita in una o più istruzioni RISC86. Le istruzioni RISC86 sono istruzioni "carica/memorizza" di tipo RISC che possono essere eseguite nelle pipeline. Dalla descrizione dell'Nx586 sembra che debba eseguire il codice ottimizzato per il Pentium a una velocità uguale o superiore rispetto al Pentium stesso.

A proposito dei chip AMD e Cyrix non si sa molto. Ma entrambi dicono di superare le prestazioni del Pentium. Sembra che il Cyrix M1 ottenga questo risultato tramite l'esecuzione speculativa.

Infine IBM sta lavorando su un PowerPC che include un decoder di istruzioni 80x86. Questa CPU, denominata PowerPC 615, dovrebbe decodificate le istruzioni 80x86 e convertirle in istruzioni native del PowerPC.

Appendice K

Glossario

Accoppiamento delle istruzioni

Il processo che consente di eseguire contemporaneamente due istruzioni in due diverse pipeline della CPU. Sul Pentium vi sono due pipeline: U e V.

AGI

Address Generation Interlock ovvero blocco di generazione degli indirizzi

Allineamento

Il posizionamento di dati o codice su determinati indirizzi (allineamento a 2, 4 o 8 byte).

ALU

Arithmetic Logic Unit ovvero unità aritmetico-logica.

ASCII

American Standard Code for Information Interchange. Un codice standard che rappresenta caratteri e simboli internazionali.

Assembler

Un programma che traduce in codice oggetto (in linguaggio macchina) un programma in linguaggio assembler.

Base

Vedere anche indirizzo base. Nei sistemi numerici è utilizzata per specificare il numero di cifre che compongono il sistema (10 cifre per il sistema in base 10, 2 per il sistema binario e 16 per il sistema esadecimale).

BCD

Binary Coded Decimal. Vedere anche Packed BCD. Si tratta di un formato per la codifica di numeri in base 10 in cui i 4 bit di ordine inferiore vengono utilizzati per memorizzare il valore numerico.

BIOS

Basic Input Output System. Software interno (normalmente contenuto in memoria ROM) che avvia il computer e controlla l'hardware come la tastiera, lo schermo e i dischi.

Bit

Una cifra binaria. Può essere uguale a 0 o a 1.

Blocco AGI (Address Generation Interlock)

Un ritardo provocato dal fatto che un registro necessario per il calcolo di un indirizzo viene calcolato da un'istruzione precedente punto si verifica su processori con pipeline (dal 486 in avanti).

Buffer di destinazione del salto

Un piccolo buffer utilizzato per conservare la "storia" dei salti per la funzione di previsione della destinazione dei salti del Pentium.

Byte

Dati costituiti da 8 bit.

Cache

Un piccolo buffer di memoria veloce che contiene una copia delle porzioni di memoria più recentemente utilizzate. Vedere anche cache del disco.

Cache del disco

Un programma e/o un'area di memoria che si occupa di conservare i dati più frequentemente richiamati dal disco in modo da accelerarne gli accessi.

Cache esterna

Una memoria cache che non si trova all'interno del chip della CPU.

Carattere

Un tipo di dati delle dimensioni di un byte.

Checksum

Un semplice schema di rilevamento degli errori in cui i valori vengono sommati in una variabile da confrontare con un valore precedentemente calcolato in modo analogo.

Chip

Un piccolo frammento di materiale semiconduttore su cui viene posizionato un circuito elettronico. Il chip della CPU è chiamato anche microprocessore.

Cicli di CPU
Impulsi periodici creati da un circuito elettronico che regola l'attività della CPU.

CISC
Complex Instruction Set Computer

Clock
La velocità alla quale un microprocessore è in grado di eseguire le istruzioni; viene misurato in Mhz.

Codice oggetto
Una forma intermedia di linguaggio macchina prodotta da assembler e compilatori; è strutturata in modo tale da consentire il linking con altri moduli in codice oggetto.

Commutazione di task
Il trasferimento dell'esecuzione da un task a un altro.

Compilatore
Un programma che traduce un programma in linguaggio ad alto livello (ad esempio il C, il Pascal o il Fortran) in linguaggio macchina e talvolta in linguaggio assembler.

Complemento a due
Un'operazione matematica in cui un valore binario viene moltiplicato per -1. Ogni bit del numero viene scambiato e quindi al valore viene sommata 1 unità.

Complex Instruction Set Computer
Computer progettati con set di istruzioni complesse e talvolta irregolari. L'architettura dell'80x86 è considerata di tipo CISC. Vedere RISC.

Coprocessore matematico
Un processore matematico per numeri in virgola mobile.

Coprocessore numerico
Un processore matematico in virgola mobile.

CP/M
Control Program for Microcomputer. Un sistema operativo originariamente progettato per computer con microprocessori 8080 e Z-80.

CPL
Livello di privilegio corrente.

CPU

Central Processing Unit. La principale unità di elaborazione del computer. Chiamata anche processore, microprocessore o chip.

CRC

Cyclic Redundancy Check. Un complesso sistema di rilevamento degli errori simile al Checksum ma in cui ogni valore viene considerato sulla base della sua posizione. Questo aumenta l'affidabilità della routine di rilevamento degli errori.

Debugger

Un programma che consente l'esecuzione, il controllo e la modifica di programmi e dati per permettere a un programmatore di individuare gli errori del programma.

Descrittore del livello di privilegio

Il livello di privilegio applicato a un segmento; si trova nel descrittore di segmento.

Descrittore di Gate

Uno speciale descrittore di segmento che costituisce la destinazione di una chiamata con un salto. Tale descrittore è utilizzato per modificare il livello di privilegio della CPU. Vi sono descrittori di quattro tipi: Call, Interrupt, Trap e Task.

Descrittore di segmento

Un elemento di una tabella di descrittori che definisce l'indirizzo iniziale, la lunghezza e gli attributi di un segmento in modalità protetta.

Direttiva

Un'istruzione assembler che contiene informazioni per il programma assembler; tali informazioni non verranno assemblate ed eseguite come parte del programma.

Disassembler

Un programma che cerca di ricostruire un file in linguaggio assembler partendo da un programma in linguaggio macchina.

DMA

Direct Memory Access ovvero accesso diretto alla memoria. Un metodo che consente a un dispositivo di trasferire dati da o verso la memoria senza impegnare la CPU.

DOS

Disk Operating System. Chiamato anche MS-DOS o PC-DOS.

DPL
Vedere descrittore del livello di privilegio.

DR-DOS
Digital Research DOS (ora Novell DOS).

DRAM
RAM dinamica. Il tipo di chip di memoria utilizzati nella maggior parte dei computer. È chiamata dinamica poiché deve subire continuamente un processo di refresh (riattivazione) per evitare che il suo contenuto venga perso.

Dword
Un tipo di dati costituito da una doppia word ovvero da 32 bit.

EA
Effective Address ovvero Indirizzo effettivo

Eccezione
Una chiamata forzata a una routine di interrupt che gestisce varie condizioni di errore.

Editor
Un programma che consente di creare e modificare i file.

Emulatore
Un programma che cerca di emulare (ovvero di funzionare come) un altro programma o una macchina.

Etichetta
Un identificatore utilizzato nel linguaggio assembler per specificare un indirizzo di memoria per nome invece che attraverso il suo effettivo indirizzo numerico.

Fault
Un'eccezione che viene richiamata inserendo nello stack l'indirizzo dell'istruzione che ha provocato l'errore.

File
Una raccolta organizzata di dati o informazioni, normalmente memorizzata su disco con un determinato nome.

Flat (modello)
Un modello di memoria in cui tutti i registri di segmento sono uguali e di dimensioni maggiori di 64 KB.

FPP
Floating-Point Processor ovvero processore in virgola mobile.

FPU
Floating Point Unit ovvero unità in virgola mobile

GB
GibaByte

GDT
Global Descriptor Table ovvero tabella dei descrittori globali.

Gestore di interrupt
Una routine progettata per rispondere a un interrupt.

GigaByte (GB)
2^{30} o 1073741824 byte.

Hertz
Una frequenza di un ciclo per secondo.

Hz
Abbreviazione di hertz.

I/O
Input/output.

IDT
Interrupt Descriptor Table ovvero tabella dei distributori di interrupt.

IEEE
Institute of Electrical and Electronic Engineers. Un'organizzazione nota per lo sviluppo di standard elettrici ed elettronici per l'industria dei computer.

Indirizzo
Locazione di un oggetto in memoria. Vedere anche Indirizzo lineare e Indirizzo fisico

Indirizzo base

L'indirizzo iniziale di una struttura o di un array.

Indirizzo effettivo

La combinazione di un registro base, un registro indice e uno scostamento utilizzato per produrre un offset all'interno del segmento.

Indirizzo fisico

L'indirizzo di memoria hardware prodotto dal microprocessore. Il massimo indirizzo fisico è determinato dal numero dei piedini che costituiscono il bus per indirizzi del microprocessore.

Indirizzo lineare

Un indirizzo a 20, 24 o 32 bit in un grande spazio di memoria non segmentato. disabilitando la paginazione (la memoria virtuale), l'indirizzo lineare è un indirizzo fisico. Quando invece è attivata la paginazione il meccanismo di paginazione traduce l'indirizzo lineare in un indirizzo fisico.

Indirizzo logico

Un segmento e un offset combinati in modo da generare un indirizzo logico. L'unità di segmentazione traduce l'indirizzo logico in un indirizzo lineare.

Intero

Un numero intero positivo, negativo o uguale a 0. Sui computer gli interi hanno un intervallo piuttosto limitato; ad esempio in un byte è possibile memorizzare valori compresi fra -128 e +127.

Intero long

Vedere Intero. Dati interi in un formato a 32 bit.

Intero senza segno

Un numero intero positivo o lo 0. Sui computer gli interi senza segno hanno un intervallo limitato. Ad esempio un intero senza segno a 8 bit può contenere valori compresi fra 0 e 255.

Interprete

Un programma che esegue un altro programma leggendone ogni istruzione e interpretando le azioni da eseguire.

IVT

Interrupt Vector Table ovvero tabella dei vettori di interrupt

KB
KiloByte

KiloByte (KB)
2^{10} ovvero 1024 byte.

LDT
Local Descriptor Table ovvero tabella dei descrittori locali.

Libreria
Una raccolta di programmi o subroutine memorizzate in formato di codice oggetto normalmente all'interno di un file .EXE.

Linguaggi di alto livello
Ad esempio il C, il BASIC, il Pascal e il Fortran.

Linguaggio assembler
Un linguaggio di programmazione che si basa sull'architettura di una determinata macchina e in cui ogni istruzione viene tradotta in un'istruzione della macchina.

Linguaggio macchina
Il codice binario che una macchina (CPU) e direttamente in grado di eseguire.

Linker
Un programma che collega uno o più file di codice oggetto per produrre un file eseguibile in formato .EXE.

Linking
Il complesso processo di unione di più file oggetto con collegamento delle subroutine e dei dati da un file a un altro file.

Livello di privilegio
Codice di protezione per ogni segmento quando il sistema si trova in modalità protetta. L'intervallo va da 0 (privilegio più elevato, codice del sistema operativo) a 3 (codice a basso privilegio, applicazioni).

Livello di privilegio corrente
Il livello di privilegio del codice in esecuzione. Si applica solo al codice operante in modalità protetta e corrisponde ai 2 bit inferiori del selettore del segmento di codice.

Maschera

Una configurazione di bit realizzata per essere unita logicamente con altri dati in modo da evidenziare alcuni bit del valore originario (si chiama così perché gli altri bit vengono mascherati).

MB

MegaByte

MegaByte (MB)

2^{20} o 1048576 byte.

Megahertz

Un milione di hertz; 1 Mhz equivale a un milione di cicli per secondo.

Memoria virtuale

Uno schema utilizzato per consentire ai programmi di allocare e utilizzare più memoria rispetto a quella disponibile sul sistema; opera spostando e scambiando fra la memoria e il disco fisso quelle porzioni di memoria che non sono più necessarie o vengono utilizzate raramente.

Mhz

Megahertz

Microprocessore

Un processore per computer completamente contenuto in un circuito integrato (o chip).

Modalità protetta

Una modalità della CPU in cui gli indirizzi di memoria sono protetti e dunque non è consentito leggere e/o scrivere su segmenti di codice senza averne l'autorizzazione.

Modalità reale

L'unica modalità della CPU disponibile sui microprocessori 8088 e 8086 e la modalità iniziale dei microprocessori dall'80286 in avanti. In modalità reale non vi è alcun sistema di protezione della memoria.

Modalità V86

Modalità virtuale 8086.

Modalità virtuale 8086

Una modalità dei microprocessori 386 e successivi che consente di emulare l'architettura 8086. Un sistema operativo può eseguire una serie di task in modalità protetta e in modalità virtuale 8086.

Modulo

Una parte di un programma, normalmente un file, contenente una o più procedure o subroutine e/o dati.

MS-DOS

Microsoft DOS

Nibble

4 bit. In un byte vi sono dunque 2 nibble.

NPX

Numerical Processor Extension ovvero estensione numerica del processore. È il nome originario del coprocessore 8087.

Offset

Un numero a 16 bit che specifica il numero di byte di distanza rispetto all'inizio di un segmento. I microprocessori dall'80386 in avanti possono avere offset a 32 bit.

Operando

Dati forniti in un registro, in memoria o immediatamente all'interno di un'istruzione e che verranno utilizzati per l'elaborazione di tale istruzione.

OS

Sistema operativo

P-system

UCSD P-system.

Packed BCD

Packed Binary Coded Decimal. Un formato di dati che memorizza una cifra decimale in ogni nibble di un byte.

Pagina

Un blocco di memoria costituito da 4 KB. Si tratta delle dimensioni del blocco di memoria utilizzato per la paginazione.

Paginazione

Un metodo di gestione della memoria, da parte di un sistema operativo, che impiega sistemi di memoria virtuale. Le pagine di memoria non utilizzate vengono salvate su disco e richiamate nel momento in cui se ne presenta la necessità.

Paragrafo

16 byte di memoria allineata.

Parità

Un sistema di rilevamento degli errori normalmente utilizzato nella comunicazione di dati che controlla la parità o meno dei bit di un pacchetto di dati.

PC-DOS

Personal Computer DOS, la versione IBM di MS-DOS.

Pipeline

Una serie di fasi che un'istruzione passa per completare la propria funzione.

Porta

Un canale o un collegamento per i dati che entrano o escono dalla CPU.

Prefisso

Uno dei codici che può essere posizionato davanti ad altre istruzioni per modificare l'azione o le condizioni standard.

Prefisso dimensionale dell'indirizzo

Un prefisso che modifica le dimensioni standard per gli indirizzi utilizzati come operandi. Gli indirizzi utilizzati come operandi possono essere a 16 o 32 bit. Il valore standard per ogni segmento di codice si trova in GDT o LDT per ogni segmento.

Prefisso dimensionale dell'operando

Un prefisso per istruzioni che consente di modificare le dimensioni standard degli operandi interi. Gli operandi possono essere valori a 8, 16 o 32 bit. L'impostazione standard per ogni segmento di codice si trova in GDT o LDT per ogni segmento.

Previsione della destinazione dei salti

Una caratteristica del Pentium che tenta di prevedere la destinazione dei salti sulla base dei risultati precedenti dell'istruzione.

Processore
Abbreviazione di microprocessore. Quella parte del computer che esegue tutte le operazioni aritmetiche, logiche e di controllo di un computer.

Processore in virgola mobile
Un processore matematico che esegue operazioni su numeri in virgola mobile. L'unità esegue operazioni aritmetiche in virgola mobile su numeri a 32, 64 e 80 bit con segno ed esponenti.

Pseudo-op
Sinonimo di direttiva assembler.

Puntatore far
Un indirizzo di memoria composto da un segmento e un offset. In modalità reale il segmento è costituito dai 16 bit superiori dell'indirizzo iniziale di un segmento (un valore a 20 bit). In modalità protetta il segmento è un selettore.

Puntatore near
Un indirizzo di memoria costituito dalla sola porzione di offset dell'indirizzo. L'offset deve essere combinato con un segmento o un selettore in uno dei registri di segmento.

Quadword
Un tipo di dati costituito da 8 byte ovvero 64 bit.

RAM
Random Access Memory ovvero memoria ad accesso diretto. La memoria di lettura e scrittura utilizzata dal computer, normalmente è costituita da chip di tipo DRAM.

Reduced Instruction Set Computer
Computer realizzati con (relativamente) poche istruzioni semplici; un'architettura di tipo "carica/salva" e un set di istruzioni regolari che ne agevolano la gestione tramite pipeline.

Registro
Un'area di memorizzazione interna della CPU. Ogni registro può eseguire direttamente determinate istruzioni.

Registro base
Un registro che contiene un indirizzo base. Normalmente BX (EBX) o BP (EBP).

Rientrante

Una proprietà di una procedura o di un programma che può essere interrotto e richiamato o rieseguito e in entrambi i casi rimane intatto e continua senza problemi la propria esecuzione.

RISC

Reduced Instruction Set Computer.

Scostamento

La parte costante di un indirizzo effettivo (EA).

Segmento

In modalità reale un segmento è una porzione di memoria specificata da un indirizzo iniziale a 20 bit in cui i primi 4 bit sono sempre uguali a 0. Può avere una lunghezza fino a 64 KB. In modalità protetta un segmento è una porzione di memoria descritta da un elemento della tabella dei descrittori.

Segmento dati

L'area di memoria indirizzabile definita dal segmento nel registro DS.

Segmento dello stato

Una porzione di memoria puntata dal registro di segmento dello stato e utilizzata per contenere lo stack di sistema.

Segmento di codice

L'area di memoria indirizzabile definita dal segmento contenuto nel registro CS.

Segmento di stato del task

Un segmento utilizzato per conservare lo stato della CPU durante una commutazione di task.

Segmento extra

L'area di memoria indirizzabile definita dal segmento contenuto nel registro ES.

Selettore

In modalità protetta è un puntatore a un descrittore di segmento. Un selettore è un valore a 16 bit utilizzato in modalità protetta al posto di un indirizzo di paragrafo nel registro di segmento.

Selettore del segmento di codice

In modalità protetta, il selettore di segmento nel registro CS.

Sistema operativo

I programmi che caricano le applicazioni, che controllano l'accesso alla memoria, ai file, alle porte di I/O e così via. Ad esempio, DOS, Windows, OS/2 e UNIX sono tutti sistemi operativi.

SRAM

RAM statica. Più veloce e stabile della memoria DRAM ma presenta un consumo superiore ed è più costosa. Normalmente è utilizzata come memoria cache.

Stack

Una struttura dati di tipo LIFO (Last In First Out) utilizzata per salvare gli indirizzi di rientro, le variabili temporanee e altre informazioni sullo stato del sistema. Un sistema può avere un qualsiasi numero di stack dei quali uno solo può essere attivo.

Stringa

Una sequenza di byte. Nel linguaggio C le stringhe devono essere concluse da un byte contenente il valore 0.

Stringa di caratteri

Un tipo di dati costituito da un array di caratteri. Normalmente è seguito da un byte uguale a zero.

Struttura dati

Uno schema di organizzazione di dati correlati.

Superscalare

Una CPU che può completare più di una istruzione per ciclo macchina.

Tabella dei descrittori

Un array di descrittori di segmento. Vi può essere una tabella di descrittori globali (GDT) e più tabelle di descrittori locali (LDT). Inoltre vi può essere una tabella dei descrittori di interrupt (IDT).

Tabella dei descrittori di interrupt

Un array di descrittori utilizzati per richiamare i gestori di interrupt e le eccezioni in modalità protetta.

Tabella dei descrittori globali

Un array di descrittori di segmento per tutti i programmi in esecuzione sul sistema. Tale tabella è controllata dal sistema operativo.

Tabella dei descrittori locali
Un array di descrittori di segmento per un singolo programma.

Tabella dei vettori di interrupt
Un array di 256 puntatori far ai gestori di interrupt. Si trova all'indirizzo 0000:0000 (in modalità reale).

Task
Un programma in esecuzione o in attesa di esecuzione in un sistema multitasking.

Trap
Un interrupt, normalmente generato come risultato di un'istruzione o una condizione vietata.

TSS
Task State Segment ovvero segmento di stato del task.

UCSD P-system
University of California San Diego P-system. Un sistema basato su linguaggi semicompilati (ovvero compilati in p-code o pseudocodice).

Unità aritmetico logica
Quella porzione della CPU che esegue operazioni intere come ADD, SUB, AND, OR e CMP.

Unità in virgola mobile
La porzione dell'80486 o del Pentium che esegue le operazioni in virgola mobile.

Uscita dal segmento
Un prefisso per un'istruzione che consente di utilizzare un registro un segmento diverso da quello contenuto nel registro di segmento.

Word
Un tipo di dati costituito da 2 byte ovvero 16 bit.

Appendice L

Prodotti menzionati

TASM e Borland C++
Borland International
1800 Green Hills Rd.
Scotts Valley, CA 95067-0001
(408) 438-5300 (800) 336-6464

Cloaking Developer's Toolkit
Helix Sofrware Company
47-09 30th Street
Long Island City, NY 11101
(718) 392-3100

Pentium Processor User's Manual
Intel Literature
P.O. Box 7641
Mt. Prospect, IL 60054-7641
(800)548-4725

Kedit (editor)
Mansfield Software Group
P.O. Box 532
Storrs, CT 06268
(203) 429-8402

MASM, Microsoft C/C++ e Visual C++
Microsoft Corp.
One Redmond Way
Redmond, WA 98052-6399
(206) 882-8080 (800) 426-9400

Soft-Ice e Bounds Checker
Nu-Mega Technologies, Inc.
P.O. Box 7780
Nashua, NJ 03060-7780
(603) 889-2386

Periscope Model IV
Periscope
1475 Peachtree St. Suite 100
Atlanta, GA 30309
(404) 888-5335 (800) 722-7006

PentOpt Professional e ASMFLOW Professional
Quantasm Corp.
19672 Stevens Creek Blvd. Suite 307
Cupertino, CA 95014-2465
(408) 244-6826 (800) 765-8086

Sourcer (disassembler)
V Communications
4320 Stevens Creek Blvd., Suite 275
Sanjose, CA 95129
(408) 296-4224

Indice analitico

88 386 0260-3 G. Harvey, *Usare Lotus 1-2-3 senza fatica*
88 386 0201-8 A. Rathbone, *Usare OS/2 senza fatica*
88 386 0211-5 J.R. Levine, M. Levine Young *Usare Unix senza fatica*
88 386 0262-X A. Rathbone, *Usare Windows 3.1 senza fatica*
88 386 0218-2 B. Slick, *Usare Word 6 per DOS senza fatica*
88 386 0192-5 D. Gookin, *Usare WordPerfect senza fatica*
88 386 0299-9 J. Walkenbach, *Usare 1-2-3 per Windows senza fatica*
88 386 0298-0 D. Gookin, *Usare Word per Windows senza fatica*
88 386 0300-6 A. Rathbone, *Gestire il computer senza fatica*
88 386 0313-8 J.R. Levine, *Usare Internet senza fatica*
88 386 0304-9 A. Rathbone, *Windows per saperne di più senza fatica*
88 386 0305-7 D. Gookin, *DOS per saperne di più senza fatica*
88 386 0306-5 K. Murray, *SOS per Windows*
88 386 0307-3 K. Murray, *SOS per DOS*

Guide rapide senza fatica

88 386 0357-X E. Finkelstein, *Guida rapida AutoCAD senza fatica*
88 386 0352-9 G. Harvey, *Guida rapida Windows 95 senza fatica*
88 386 0339-1 J.R. Levine, *Guida rapida Unix senza fatica*
88 386 0338-3 M.J. Partington, *Guida rapida Works 3 senza fatica*
88 386 0337-5 J.R. Levine, *Guida rapida Internet senza fatica*
88 386 0217-4 G. Harvey, *Comandi DOS 6 senza fatica*
88 386 0269-7 J. Walkenbach, *Guida rapida Excel senza fatica*
88 386 0301-4 G. Harvey, *Guida rapida WordPerfect per Windows senza fatica*
88 386 0302-2 G. Linch, *Guida rapida Word per Windows senza fatica*
88 386 0303-0 S. Stuple, *Guida rapida Access per Windows senza fatica*
88 386 0268-9 J. Walkenbach, *Comandi Lotus 1-2-3 senza fatica*
88 386 0267-0 J. Walkenbach, *Comandi Windows senza fatica*

Microcalcolatori

88 386 0371-5 L. Lamb, J. Peek, *E-mail a portata di mano*
88 386 0362-6 J. Lammers, *Come usare 3D Studio 4*
88 386 0356-1 J.F. Hughes, *Come usare Novell Netware 4.1*
88 386 0354-5 R.A. Neibauer, *Come usare Word per Windows 95*
88 386 0358-8 M.S. Matthews, *Come usare Excel per Windows 95*
88 386 0353-7 L. Lemay, *Il manuale HTML*
88 386 0348-0 J. Manger, *Come usare NetScape e WWW*
88 386 0335-9 J.D. Hood, *Come usare AutoCAD LT*
88 386 0334-0 T. Sheldon, *Come usare Windows 95*
88 386 0330-8 S. Rimmer, *Pianeta Internet*
88 386 0329-4 D.P. Dern, *Alla scoperta di Internet*
88 386 0322-7 P.M. Ridge, D.M. Golden, I. Luk, S.E. Sindorf, *Come usare Sound Blaster*
88 386 0310-3 M. Davis, *Microsoft Office Applicazioni integrate*
88 386 0314-6 M.S. Matthews, *Come usare CorelDRAW! 5*
88 386 0319-7 R. Mansfield, *Come usare Visual Basic for Applications*
88 386 0321-9 A. Droblas, S. Greenberg, *Come usare Photoshop 3*

88 386 0325-1 K.V. Reed, *Comunicare via modem (libro + 2 dischi)*
88 386 0309-X M. Davis, *Come usare Access 2.0*
88 386 0316-2 P. Sheldon, *Come usare dBASE per Windows*
88 386 0295-6 T. Badgett, C. Sandler, *Progetti multimediali*
88 386 0292-1 M. Matthews, *Come usare Excel 5 per Windows*
88 386 0296-4 E. Forsan, *Come usare 3D Studio3.0*
88 386 0297-2 D.Hakala, *Come usare il modem*
88 386 0278-6 P. Hoffman, *Come usare Word 6 per Windows*
88 386 0293-X The Cobb Group, *Guida a Word 6 per Windows*
88 386 0166-6 E.Ihrig, M.S. Matthews, *Come usare CorelDRAW! 3*
88 386 0263-8 E. Jones, *Come usare FoxPro 2.5 per Windows*
88 386 0265-4 E. Forsans, *Come usare 3D Studio*
88 386 0205-0 M. Davis, *Come usare Microsoft Access*
88 386 0207-7 A. Hassadi, G. Gruman, *Come usare QuarkXPress*
88 386 0210-7 G. Cornell, *Come usare Visual Basic 2*
88 386 0270-0 P. Hoffman, *Come usare Word 5.1 per Macintosh*
88 386 0266-2 P. Hoffman, *Come usare Word 6 per DOS*
88 386 0206-9 H. Schildt, *Manuale MS-DOS 6*
88 386 0204-2 P. Shaddock, *Multimedia in pratica*
88 386 0264-6 J. Groves, *Window NT le risposte*
88 386 0258-1 E. Ihrig, M.S. Matthews, *Come usare CorelDRAW! 2*
88 386 0168-2 M. Campbell, *Come usare Lotus 1-2-3 per Windows*
88 386 0234-4 R. Soucie, *Come usare Microsoft Excel 3 per Windows*
88 386 0190-9 R. Soucie, *Come usare Microsoft Excel 4 per Windows*
88 386 0232-8 J. Rampa, *Come usare Microsoft Word 5.5*
88 386 0254-9 C. Townsend, *Come usare MS-DOS 5 e MS-DOS Shell*
88 386 0240-9 E. Jones, *Come usare Paradox 3.5 Versione italiana*
88 386 0191-7 L. Biow, *Come usare Quattro Pro 3*
88 386 0237-9 E. Ermacora, *Come usare Visual Basic*
88 386 0231-X T. Sheldon, *Come usare Windows 3.1*
88 386 0248-4 P. Hoffman, *Come usare Word 2 per Windows*
88 386 0067-8 W.A. Ettlin, *Come usare WordsStar 6.0 Versione italiana*
88 386 0243-3 L. Poole, *Guida a System 7*
88 386 0131-3 L.L. Lorenz, R.M. O'Mara, *Guida a Windows 3.1*
88 386 0253-0 M.W. Crane, *Guida a Word 2 per Windows*
88 386 0188-7 J. Heid, *Guida MacWord a Macintosh*
88 386 0257-3 The Wait Group, *MS-DOS QBASIC Guida del programmatore*
88 386 0226-3 E. Jones, *Come usare dBASE IV 1.1*
88 386 0225-5 P. Fezzi, R. Rocchetti, *Come usare Framework III e IV*
88 386 0174-7 L.J. Scanlon, *Come usare WordPerfect Versione 5.1 italiana*
88 386 0203-4 A. Comi, *Grafica matematica con il personal computer. Biomorfi e frattali (libro + disco)*
88 386 0223-9 L.L. Lorenz, R.M. O'Mara, *Guida a Windows 3*
88 386 0200-X M.W. Crane, *Guida a Word per Windows*
88 386 0194-1 J.C. Dvorak, N. Anis, *Guida Dvorak al DOS e alle prestazioni del PC*
88 386 0062-7 C.H. Pappas, W.H. Murray III, *Manuale 80836*
88 386 0239-5 H. Schildt, *Manuale MS-DOS 5*

88 386 0177-1 R. Evans, *Manuale Norton Utilities Versione 4.5 inglese*
88 386 0176-3 C. Ackerman, *Manuale PC Tools Deluxe Versione 6.0 inglese*
88 386 0221-2 C. Rubin, *Microsoft Works II*
88 386 0172-0 D. Innman,B. Albrecht, *Programmare in QuickBASIC Versione 4.5*
88 386 0151-8 H. Shildt, *Programmare in Turbo C++*
88 386 0149-6 G.M. Perry, *Come usare Microsoft Word 5*
88 386 0148-8 M.S. Matthews, B.C. Matthews, *Come usare WordsStar 5.5*
88 386 0152-6 A. Comi, *Elaborazioni grafiche con il personal computer*
88 386 0126-7 L. Biow, *Introduzione a Quattro*
88 386 0164-X A.T. Williams, *Lotus1-2-3 Release 2.2 Versione inglese*
88 386 0158-5 C. Siechert, C. Wood, *Manuale MS-DOS 4*
88 386 0159-3 C. Siechert, C. Wood, *Manuale MS-DOS 3.30*
88 386 0184-4 D. Weber, *Manuale Novell Netware*
88 386 0171-2 T. Sheldon, *Manuale Windows 3*
88 386 0093-7 J. D. Carrabis, *dBASE III PLUS Applicazioni in rete locale*
88 386 0085-6 F. Baeseler, B. Heck, *Introduzione al Desktop Publishing*
88 386 0068-6 M.S. Matthews, B.C. Matthews, *PageMaker 3.0 per Personal MS-DOS*
88 386 0146-1 M. Guerriero, H. Zampariolo, *Programmare in Fred con Framework II e III*
88 386 0077-5 F.E. Mosher, D.I. Schneider, *Turbo BASIC Elementi di programmazione*
88 386 0076-7 S. Wood, *Guida al Turbo Pascal 4.0*
88 386 0053-8 C.W. Kid, *Lotus 1-2-3: applicazioni finanziari*
88 386 0071-6 H. Schildt, *Turbo C Elementi di programmazione*
88 386 0073-2 H. Schildt, *Turbo C Programmazione avanzata*
88 386 0069-4 H. Schildt, *Turbo Prolog 1.1 Programmazione avanzata*
88 386 0049-X W.H. Murray, C.H. Pappas, *Assembler per l'80286/80386*
88 386 0056-2 Z. Jones, *Come usare dBASE III PLUS*
88 386 0064-3 W. Ettlin, *Come usare WordStar 4.0*
88 386 0050-3 D. Andersen, C. Cooper, B. Dempsey, *dBASE III in pratica*
88 386 0047-3 C. Siechert, C. Wood, *Manuale MS-DOS 3.20*
88 386 0051-1 D.W. Carroll, *Programmazione in Turbo Pascal*
88 386 0060-0 P.R. Robinson, *Programmazione in Turbo Prolog*
88 386 0019-8 E.M. Baras, *Come usare il Symphony*
88 386 0043-0 T.J. Byers, *Guida al PC AT IBM*
88 386 0044-9 D.A. Kater, R.L. Kater, *Guida alle stampanti Epson*
88 386 0041-4 W. Ettin, G. Solgerg, *GW-BASIC per Personal Computer Olivetti*
88 386 0037-6 D. Kruglinski, *Introduzione al Framework*
88 386 0042-2 D. Watt, *Logo per il Commodore 64*
88 386 0052-X P. Hoffman, T. Nicoloff, *Manuale MS-DOS per PC Olivetti*
88 386 0048-1 H. Peckham, W. Ellis Jr, E. Lodi, *Programmazione strutturata in BASIC*
88 386 0026-0 W. Ettlin, *Come usare il multiplan*
88 386 0030-9 W. Ettlin, G. Solberg, *BASIC Microsoft*
88 386 0021-X L.J. Graham, T. Field, *Guida al PC-IBM*
88 386 0025-2 C. Morgan, M. Waite, *Manuale 8086/8088*
88 386 0008-2 H. Peckham, *BASIC e il PC-IBM in pratica*
88 386 0001-5 J. Heilborn, R. Talbott, *Guida al Commodore 64*

Informatica professionale

88 386 0328-6 G. Cornell, *Il manuale Visual Basic 4*
88 386 0372-3 W.H. Murray, C.H. Pappas, *Il manuale Visual C++ Programmazione Windows*
88 386 0345-6 W.H. Murray, C.H. Pappas, *Il Manuale Visual C++ Il linguaggio*
88 386 0365-0 J.D. Schank, *Il manuale Client/Server*
88 386 0347-2 C. Ganz, *Il manuale CA-Visual Object*
88 386 0346-4 U. Black, *Il manuale TCP/IP*
88 386 0342-1 J.H. Zirbel, *AutoCAD 13 I fondamenti*
88 386 0344-8 J.H. Zirbel, *AutoCAD 13 Uso avanzato*
88 386 0343-X S. Holzner, *Il manuale OLE 2.0*
88 386 0336-7 M.J. Corey, *Il manuale Oracle*
88 386 0332-4 H. Schildt, *Windows 95 Programmazione in C e C++*
88 386 0323-5 A. Schulman, *Windows 95 dentro il sistema*
88 386 0308-1 W.H. Murray, C.H. Pappas, *Il manuale Borland C++*
88 386 3401-7 R. Adinolfi, *Reti di computer*
88 386 0196-8 F. Brusca, D. Ronzitti, R. Smoquina, *Da Clipper a C++*
88 386 0230-1 S. Straley, *Manuale Clipper 5.2*
88 386 0271-9 S. Cannan, G. Otten, *Manuale SQL*
88 386 0275-1 D.J. Kruglinski, *Manuale Visual C++*
88 386 0274-3 S. Rimmer, *Windows Bit-Mapped Graphics*
88 386 0175-5 H. Schildt, *Arte della programmazione in C*
88 386 0214-X Que Development Group, *AutoCAD 12 I Fondamenti*
88 386 0193-3 Que Development Group, *AutoCAD 12 Uso avanzato*
88 386 0236-0 S. Rimmer, *Bit Mapped Graphics*
88 386 0136-4 B. Eckel, *Programmare in C++*
88 386 0185-2 M.G. Naugle, *Reti locali*
88 386 0180-1 C. Halliday, *Segreti del PC*
88 386 0227-1 R.D. Ainsbury, *Segreti di DOS 6*
88 386 0202-6 M. Salin, *Applicazioni statiche con SPSS Versione 4.01*
88 386 0259-X R. Spence, *Programmare in Clipper Versione 5.01, Seconda edizione*
88 386 0173-9 R. Rocchetti, A. Moroni, *Programmazione in Excel 4 e Q+E per Windows*
88 386 0256-5 R. Salcedo, *Programmare in Paradox 3.5*
88 386 0199-2 D. Ince, *Programmazione a oggetti in C++*
88 386 0241-7 B. Livingston, *Segreti di Windows 3.1*
88 386 0238-7 P. Sanasi, *Sistema informativo in rete*
88 386 0220-4 J. Occhiogrosso, *Clipper 5.0 Le librerie (libro + disco)*
88 386 0169-0 P.A. Darnel, P.E. Margolis, *C Manuale di programmazione*
88 386 0186-0 E. Teja, L. Johnson, *Computer Graphics con PC IBM e PS/2 Hardware e software*
88 386 0224-7 R. Pressman, *Principi di ingegneria del software*
88 386 0160-7 Price Waterhouse, *Computer Virus*
88 386 0153-4 D.M. Kalman, *Manuale dei linguaggi dBASE*
88 386 0165-8 K.W. Christopher, B.A. Feigenbaum, S.O. Saliga, *MS-DOS Manuale di programmazione*
88 386 0154-2 R. Spence, *Programmare in Clipper*
88 386 0059-7 M. Liskin, *Programmazione avanzata in dBASE III Plus*

Guide complete

88 386 0351-0 H. Schildt, *Guida completa C++*
88 386 0340-5 H. Schildt, *Guida completa C 2ed.*
88 386 0311-1 M. Matthews, S.Seymour, *Guida completa Excel 5*
88 386 0212-3 K. Jamsa, *Guida completa DOS 6*
88 386 0140-2 M. Matthews, S.Seymour, *Guida completa Excel 4 per Windows*
88 386 0215-8 O'Brien, *Guida completa Turbo Pascal 7*
88 386 0242-5 T. Sheldon, *Guida completa Windows Versione 3.1*
88 386 0229-8 N. Johnson, *Guida completa AutoCAD Versione 11 inglese*
88 386 0228-X K. Jamsa, *Guida completa DOS 5*
88 386 0255-7 S.K. O'Brien, *Guida completa Turbo Pascal Versione 6 inglese*
88 386 0183-6 H. Schildt, *Guida completa C ANSI e C++*
88 386 0182-8 G.T. Le Blond, W.B. Le Blond, B. Heslop, *Guida completa dBASE IV*
88 386 0163-1 M. Campbell, *Guida completa Lotus 1-2-3 Versione 3 inglese*
88 386 0181-X S. Coffin, *Guida completa UNIX System V Release IV*
88 386 0187-9 T. Scheldon, *Guida completa Windows Versione 3.0*
88 386 0156-9 N. Johnson, *Guida completa AutoCAD Versione 10 inglese*
88 386 0147-X K. Jamsa, *Guida completa DOS Versione 3.30*
88 386 0161-5 M. Campbell, *Guida completa Lotus 1-2-3 Versione 2.2 iglese*
88 386 0162-3 S.K. O'Brien, *Guida completa Turbo Pascal Versione 5.5 inglese*
88 386 0066-X J.D. Carrabis, *Guida completa dBASE III PLUS*
88 386 0065-1 M. Campbell, *Guida completa Lotus 1-2-3*
88 386 0124-0 H. Schildt, *Guida completa Turbo C*
88 386 0078-3 S.K. O'Brien, *Guida completa Turbo Pascal*
88 386 0081-3 K. Jamsa, *Guida completa DOS*

Guide tascabili

88 386 0246-8 V. Wolverton, *Guida rapida Hard Disk Tecniche di gestione*
88 386 0252-2 P. Rinearson, *Guida rapida Microsoft Word 5.5*
88 386 0244-1 V. Wolwerton, *Guida rapida MS-DOS 5*
88 386 0245-X K. Jamsa, *Guida rapida MS-DOS batch file*
88 386 0247-6 K. Jamsa, *Guida rapida MS-DOS QBasic*
88 386 0251-4 J.L. Viescas, *Guida rapida Norton Utilities versione 5*
88 386 0250-6 C. Townsend, *Guida rapida PCTools Versione 6*
88 386 0178-X M. Liskin, *Guida tascabile dBASE IV*
88 386 0179-8 M. Campbell, *Guida tascabile Lotus 1-2-3 Versione 2 e 3 inglesi*
88 386 0122-4 C. Gilbert, *Guida tascabile WordStar Versione 5.5*
88 386 0105-4 H. Schildt, *Guida completa Turbo C*
88 386 0104-6 K. Jamsa, *Guida tascabile Turbo Pascal 4.0*
88 386 0100-3 H. Schildt, *Guida tascabile C*
88 386 0089-9 M. Liskin, *Guida tascabile dBASE III Plus*
88 386 0102-X K. Jamsa, *Guida tascabile DOS Versione 3.30*
88 386 0098-8 M. Campbell, *Guida tascabile Lotus 1-2-3*

Ipertesti

88 386 0930-6 C. De Francesco, *Iperlibro*
88 386 0918-7 C. De Francesco, *IperPC*

Istruzione scientifica

88 386 0703-6 S. Ceri, D. Mandrioli, L. Sbattella, *Informatica: istituzioni. Linguaggio di riferimento Ansi C*
88 386 0692-7 P. Maestrini, *Sistemi operativi*
88 386 0654-4 P. Demichelis, E. Piccolo, *Introduzione all'informatica*
88 386 0668-4 S. Ceri, D. Mandrioli, L. Sbattella, *Istituzioni di informatica. Linguaggio di riferimento Pascal*
88 386 0631-5 H. Horowitz, S. Anderson Feed, *Strutture dati in C*
88 386 0635-8 E. Rich, K. knight, *Intelligenza artificiale (2/e)*
88 386 0641-2 J.D. Musa, A. Jannino, K. Okumoto, *Affidabilità del software*
88 386 0655-2 V. Comincioli, *FORTRAN 77 Introduzione e applicazioni numeriche*
88 386 0645-5 G. Ausiello, C. Batini, D. Mandrioli, M. Protasi, *Modelli e linguaggi dell'informatica*
88 386 0224-7 R. Pressman, *Principi di ingegneria del software*
88 386 0643-9 D.W. Rolston, *Sistemi esperti*
88 386 0648-X G. Pelagatti, *Sistemi di elaborazione*
88 386 0637-4 M. Milénkovic, *Sistemi operativi*
88 386 0624-2 M.E. Mortenson, *Modelli geometrici in computer graphics*
88 386 0617-X K.S. Fu, R.C. Gonzales, C.S.G. Lee, *Robotica*
88 386 0636-6 D. Mandrioli *Elementi di informatica*
88 386 0613-7 R. Morgan, H. McGilton, *Il sistema operativo UNIX System V*
88 386 0608-0 R. Thomas, L.R. Rogers, J.L. Yates, *UNIX System V complementi di programmazione*
88 386 0615-3 J.W.L.Ogilvie, *Linguaggio Modula-2*
88 386 0606-4 W.M. Newmann, R.F. Sproull, *Principi di computer graphics*
88 386 0607-2 L. Hancock, M. Krieger, *Linguaggio C*
88 386 0605-6 M.L. Schagrin, W.J. Rapaport, R.R. Dipert, *Logica e computer*
88 386 0603-X H. McGilton, R. Morgan, *Il sistema operativo UNIX*
88 386 0601-3 S. Harrington, *Computer graphics Corso di programmazione*

Il Manuale Pentium
Ai programmatori C/C++ e assembler è illustrato come migliorare le prestazioni dei programmi sfruttando i vantaggi dell'architettura superscalare della CPU Pentium™ Intel® per aumentare le performance dal 100% al 400%.
Il dischetto contiene il programma di ottimizzazione per il Pentium, il debugger in modalità protetta a 32 bit con supporto DPMI e il software multifunzionale di valutazione delle prestazioni del codice a 16 e 32 bit.
Lire 49.000 (i.i.)
ISBN 88-386-0367-7
9 788838 603679

ip

Il Manuale Pentium

0367-7